2

RL-TR-91-274, Vol Vc (of five)
Final Technical Report
November 1991

ROME LABORATORY

PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Penelope Tutorial

ORA Corporation

Sponsored by
Strategic Defense Initiative Office

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RL-TR-91-274, Vol Vc (of five) has been reviewed and is approved for publication.

APPROVED:

John C. Faust

JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:

[illegible]

RAYMOND P. URTZ, JR.
Director
Command, Control and Communications Directorate

PENELOPE: AN ADA VERIFICATION ENVIRONMENT,
Penelope User Guide: Penelope Tutorial

Geoffrey Hird

Contractor:	ORA Corporation
Contract Number:	F30602-86-C-0071
Effective Date of Contract:	19 Aug 86
Contract Expiration Date:	30 Sep 89
Short Title of Work:	PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Penelope Tutorial
Period of Work Covered:	Aug 86 - Aug 89
Principal Investigator:	Maureen Stillman
Phone:	(607) 277-2020
RL Project Engineer:	John C. Faust
Phone:	(315) 330-3241

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by John C. Faust, RL/C3AB, Griffiss AFB 13441-5700, under contract F30602-86-C-0071.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE November 1991	3. REPORT TYPE AND DATES COVERED Final Aug 86 - Aug 89

4. TITLE AND SUBTITLE
PENELOPE: AN ADA VERIFICATION ENVIRONMENT,
Penelope User Guide: Penelope Tutorial

5. FUNDING NUMBERS
C - F30602-86-C-0071
PE - 35167G/63223C
PR - 1070/B413
TA - 01/03
WU - 02

6. AUTHOR(S)
Geoffrey Hird

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ORA Corporation
301A Dates Drive
Ithaca NY 14850-1313

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Strategic Defense Initiative
Office, Office of the
Secretary of Defense
Wash DC 20301-7100

Rome Laboratory (C3AB)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-91-274,
Vol Vc (of five)

11. SUPPLEMENTARY NOTES
RL Project Engineer: John C. Faust/C3AB/(315) 330-3241

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This tutorial introduces program verification with Penelope and Penelope's proof editor. Penelope incrementally generates verification conditions for a program, and the proof of those verification conditions implies that the program meets its specifications. Penelope's proof editor supports proofs of the verification conditions in ordinary first-order logic.

We present the proof rules for Penelope, and discuss specification and proof strategies by means of examples.

14. SUBJECT TERMS
Ada, Larch, Larch/Ada, Formal Methods, Formal Specification, Program Verification, Predicate Transformers, Ada Verification

15. NUMBER OF PAGES
110

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

Chapter 1

Introduction

This tutorial is intended to show the user how to use Penelope to construct verified Ada programs. It is not assumed that the reader is an expert in logic or program verification, but rather that he is from a computer science background. However, a certain amount of previous exposure to the concepts is assumed. The reader should be familiar with the basics of the standard Floyd/Hoare style of verification. One introduction to this topic is Gries's book [2]. We also assume some familiarity with the basics of formal logic and proof systems. Most of the experience with formal logic languages can be picked from Gries's book, though a more comprehensive treatment can be found in any number of logic texts, such as Mendelson [3]. The proof system for the mathematical logic component of Penelope is a Gentzen style sequent calculus, which is discussed in [5]. Lastly, we assume that the reader has read the Guide to Penelope [4].

The Guide to Penelope shows how to write a program, enter traits, assertions and invariants, and generate verification conditions (VCs). It shows this in a syntactic sense—what commands you must give to enter an assertion at a specific point, which mouse button to click on to generate a VC, etc.. This tutorial continues from there. We proceed by a series of worked examples, which together illustrate all of the main features of Penelope. Sometimes, when the whole of a Penelope display is reproduced, we expect the reader to make a parallel session and try various inputs, but the treatment has been designed so that the lazy reader can get a pretty good idea of what's going on just by examining the reproductions.

Details of the exact subset of Ada which is implemented in the current version of Penelope can be found in [1].

Chapter 2

Some Simple Examples with No Loops

Our subprograms have `IN` conditions and `OUT` conditions, corresponding to the usual pre- and post-conditions. A function subprogram can have `OUT` conditions, but typically has a `RETURN` condition, and this gives the value that the function is meant to return. Subprograms which propagate exceptions also have special constructs for specifying what happens in the case of exceptional termination. Details of the specification language are in [1], though the reader will pick most of it up from reading this tutorial.

We will sketch the way in which our VCs are computed. The postcondition is propagated back up through every Ada construct, undergoing predicate transformation on the way. When it reaches the top, we have (roughly) the weakest precondition. The VC then is the statement that the original precondition implies this new statement. In the case where no `IN` condition is stipulated, the precondition is evaluated to `TRUE` (it is an empty conjunction)—in this case, the VC is equivalent to the calculated weakest precondition. Similarly, if there is no stipulated `OUT` condition, the postcondition is `TRUE`.

To illustrate the basics of verification with Penelope, we start with a simple program. This program is a procedure which takes two variables and swaps the values stored in them.

```
--> 1 VCs NOT SHOWN!
PROCEDURE swap(x, y : IN OUT integer)
  --> GLOBAL ();
  --| WHERE
  --|      OUT (x=IN y);
  --|      OUT (y=IN x);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  temp : integer;

BEGIN
  temp:=x;
  x:=y;
  y:=temp;
END swap;
```

The OUT condition of the procedure has been specified. In order to be able to refer to the initial value of a variable from the point of view of a later time in the execution, we have the IN feature. Think of (IN x) as a variable which has the value which x had when execution of the procedure commenced. At the beginning of a procedure it is always true that x=IN x. Note that IN can also be applied to a whole expression. The reader should reproduce the above in an actual Penelope session and generate the pre- and postconditions for all of the statements in the procedure to observe how the transformations progress backwards from the OUT condition, which is the postcondition for the whole program. We obtain the following.

```
--> 1 VCs NOT SHOWN!
PROCEDURE swap(x, y : IN OUT integer)
  --> GLOBAL ();
  --| WHERE
  --|      OUT (x=IN y);
  --|      OUT (y=IN x);
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS
  temp : integer;

BEGIN
  --: PRECONDITION = ((y=IN y) AND (x=IN x));
  temp:=x;
  --: PRECONDITION = ((y=IN y) AND (temp=IN x));
  x:=y;
  --: PRECONDITION = ((x=IN y) AND (temp=IN x));
  y:=temp;
  --: PRECONDITION = ((x=IN y) AND (y=IN x));
END swap;
```

The last precondition is the same as the program postcondition; it is written as a "precondition" of the demarcation between the outside and the inside of the program body. A typical precondition, for example between the second and third lines of the body, doubles as the precondition of the following statement and the postcondition of the previous statement.

The reader should highlight <VC status>, select <show VC> and see that the VC was so simple that it was automatically reduced to TRUE, and thus is pronounced proved.

Our next example is a function which takes two arguments and returns the larger (or the common value if they are equal). The VC has been displayed.

```
FUNCTION max(x, y : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|       IN ((x>=0) AND (y>=0));
  --|       RETURN (IF (y<=x) THEN x ELSE y);
  --| END WHERE;
  --! VC Status: ** not proved **
  --! 1. (x>=0)
  --! 2. (y>=0)
  --! >> (IF (x<y) THEN ((y<=x)->(y=x)) ELSE ((y<=x) OR (x=y)))
  --! <proof>

IS


BEGIN
  IF (x<y) THEN
    RETURN  y;
  ELSE
    RETURN  x;
  END IF;
END max;
```

For expository purposes, we have only required that the function work for non-negative inputs, though of course it can be shown to work for all. Observe that the specification of what the function returns uses a by-cases format to define a value.

Below, we have reproduced the function body with all preconditions displayed. This is to show how the return statements are dealt with.

```
BEGIN
  --: PRECONDITION = (IF (x<y) THEN ((y<=x)->(y=x)) ELSE ((y<=x) OR (x=y)));
  IF (x<y) THEN
    --: PRECONDITION = (IF (y<=x) THEN (y=x) ELSE (y=y));
    RETURN  y;
    --: PRECONDITION = FALSE;
  ELSE
    --: PRECONDITION = (IF (y<=x) THEN (x=x) ELSE (x=y));
    RETURN  x;
    --: PRECONDITION = FALSE;
  END IF;
  --: PRECONDITION = FALSE;
END max;
```

Beginning at the bottom and moving upwards, the preconditions are FALSE at all points until we pass the first return statement.

Finally, in this chapter, we make some general remarks about our proof system. Observe the VC. It can be thought of in the following fashion: Assuming the numbered hypotheses, the conclusion following >> is true. In fact, although this is the way to think of a VC when studying it to see whether it is true or not, it is misleading. Strictly, there are no "hypotheses" or "conclusions" —in a sequent calculus, these are built right into the thing we are trying to prove. The whole of

```
--! 1. (x>=0)
--! 2. (y>=0)
--! >> (IF (x<y) THEN ((y<=x)->(y=x)) ELSE ((y<=x) OR (x=y)))
```

is a single unit, called a *sequent* in our logical system. This must be understood if one is to understand how the proof editor works. The whole sequent is the *goal* of our proof. Gentzen called the list of terms to the left of >> the *antecedent* and the term to the right of >> the *succedent*. This terminology makes for heavy-going, though, and we will be a little lax and call them the *hypotheses* and *conclusion* respectively. The reader should bear in mind that this only makes sense locally while thinking about a single goal; when we generate a subgoal, the "hypotheses" may change completely.

If we highlight the VC and hold the right mouse button down we see the options which the prover gives us. Select <SDVS-simplify> and this powerful component finishes off the proof.

Chapter 3

The Proof Editor

The proof editor is what we use to construct the proof of a VC. The logic is described fully in [5], so we will just make some brief remarks here. Proofs are usually thought of as systems wherein we keep building new valid statements out of old ones until we have the statement we want. Thus, a proof in most systems, including our sequent calculus, is a tree whose leaves are all axioms, and whose other nodes follow from these and the inference rules. The root of the tree is the final statement proved, and was our "goal" all along. When implementing a theorem prover, it often turns out to be convenient to begin with the goal (VC in our case) and work back towards the axioms. From our goal, we generate one or more subgoals, such that the goal is a consequence of the subgoals according to the inference rules. We repeat the procedure on each of the subgoals (each of which becomes in turn our "current" goal), and so on, until we reach the axioms. At this stage, the original goal is pronounced proved.

In the prover, we have a set of options for generating subgoals. Each of these options is justified by corresponding to one or more applications of the primitive inference rules of the logical system. The set is richer than strictly needed, and contains many options to make proving easier.

We will describe now the basic structure of a proof, by going through the various stages. The reader is not expected to understand what the chosen options (inference rules) do at this stage. Suppose that we are verifying a program, and after doing <show VC> we get the following.

```
--! VC Status: ** not proved **
--! 1. (n>=1)
--! >> (IF (1<=(n-2)) THEN ((2=f(3)) AND ((1=f(2)) AND (1=f(1))))
        ELSE (1=f(n)))
--! <proof>
```

Next, assuming that we have already defined the axioms bc1 and bc2 in a trait, we select <axiom>. This enables us to get the following.

```
--! VC Status: ** not proved **
--! BY axiom bc1 in trait white
--! | 1. (n>=1)
--! | 2. (f(1)=1)
--! | >> (IF (1<=(n-2)) THEN ((2=f(3)) AND ((1=f(2)) AND (1=f(1))))
          ELSE (1=f(n)))
--! | <proof>
--! □
```

We have reduced the proof of our VC to the proof of the sequent shown, by virtue of the proof-rule <axiom>. Put another way, in place of our original goal, we now have a subgoal. Two more successive selections of <axiom>, followed by a selection of <forall-anal>, gives the following.

```
--! VC Status: ** not proved **
--! BY axiom bc1 in trait white
--! | BY axiom bc2 in trait white
--! | | BY axiom istep in trait white
--! | | | BY analysis of FORALL, in 4 with 3
--! | | | | 1. (n>=1)
--! | | | | 2. (f(1)=1)
--! | | | | 3. (f(2)=1)
--! | | | | 4. (FORALL k::((k>2)->(f(k)=(f((k-2))+f((k-1))))))
--! | | | | 5. ((3>2)->(f(3)=(f((3-2))+f((3-1)))))
--! | | | | >> (IF (1<=(n-2)) THEN ((2=f(3)) AND ((1=f(2)) AND (1=f(1))))
                ELSE (1=f(n)))
--! | | | | <proof>
--! | | | □
--! | | □
--! | □
--! □
```

At this stage we have reduced the proof of the VC to the proof of the sequent which is at the right hand end of the cone pointing to the right. If now we select <imp-anal> we get the following.

```
--! VC Status: ** not proved **
--! BY axiom bc1 in trait white
--! | BY axiom bc2 in trait white
--! | | BY axiom istep in trait white
--! | | | BY analysis of FORALL, in 4 with 3
--! | | | | BY analysis of IMPLIES, in 5
--! | | | | | 1. (n>=1)
--! | | | | | 2. (f(1)=1)
--! | | | | | 3. (f(2)=1)
--! | | | | | 4. (FORALL k::((k>2)->(f(k)=(f((k-2))+f((k-1))))))
--! | | | | | >> (3>2)
--! | | | | | <proof>
--! | | | | □
--! | | | | | 1. (n>=1)
--! | | | | | 2. (f(1)=1)
--! | | | | | 3. (f(2)=1)
--! | | | | | 4. (FORALL k::((k>2)->(f(k)=(f((k-2))+f((k-1))))))
--! | | | | | 5. (f(3)=(f((3-2))+f((3-1))))
--! | | | | | 6. (3>2)
--! | | | | | >> (IF (1<=(n-2)) THEN ((2=f(3)) AND ((1=f(2)) AND (1=f(1)))
                  ELSE (1=f(n)))
--! | | | | | <proof>
--! | | | | □
--! | | | □
--! | | □
--! | □
--! □
```

Observe that this has given us two subgoals. Both of these must be proved separately in order for the proof to be complete. Each branch can be highlighted separately and dealt with. Suppose now that we are not happy with our choice of <imp-anal>, as the first subgoal is unprovable. We highlight the last selection, `BY analysis of IMPLIES, in 5`, and do <ctrl-k> to kill this last step. This takes us back to the stage immediately before the selection of <imp-anal>. This method can be used to take us back to any previous stage of the proof in one step, as highlighting any particular selection also highlights all later steps in the proof. Now we select SDVS-simplifier, which gives us the following

```
--! VC Status: ** not proved **
--! BY axiom bc1 in trait white
--! | BY axiom bc2 in trait white
--! | | BY axiom istep in trait white
--! | | | BY analysis of FORALL, in 4 with 3
--! | | | | BY SDVS simplification
--! | | | | | 1. (0<n)
--! | | | | | 2. (f(1)=1)
--! | | | | | 3. (f(2)=1)
--! | | | | | 4. (FORALL k::((3<=k)->(f(k)=(f((k-1))+f((k-2))))))
--! | | | | | 5. (f(3)=2)
--! | | | | | >> ((3<=n) OR (1=f(n)))
--! | | | | | <proof>
--! | | | | []
--! | | | []
--! | | []
--! | []
--! []
```

A couple more simple steps will reduce the goal to TRUE, which means that our proof is complete.

Before Penelope presents the VC for the first time, and before it presents the subgoals after an application of a proof-rule, it applies automatically a number of rules (e.g. those in <arithmetic>) to make the user's task easier. For this reason, the subgoals may not appear exactly as the user expects, but they will always be closely related. This is why a finished, successful proof will always end with the final goal being the axiom TRUE.

The reader should note here something which was described in [4]. If the user edits the display so that what he or she has entered is now different from what it was, the system immediately adjusts so that the whole display is entirely consistent. This is extremely useful, as it often happens that there are whole blocks which are correct, and we only wish to alter some entry in the middle of a section. For example, suppose that we have a proof which is so far 10 proof-rule selections long. We may delete selections 6-10 and save them in the kill buffer (use <ctrl-w>), delete an unwanted selection 5 (use <ctrl-k>) and finally restore our saved selections in the position after selection 4 (use <ctrl-y>). Penelope now re-processes all necessary entries and what we end up with is exactly the same as if the proof selections had originally been made in the same order as they appear in now.

The proof editor has many features designed to handle the various situations which arise. In later chapters we will be verifying different types of program, and in each case be using whatever parts of the prover are needed. In order to give a systematic and efficient exposition, we will devote this entire chapter to the prover. We will use a trivial program which in itself is of no importance, and by choosing various IN and OUT conditions we will construct VCs on which to demonstrate the prover.

We will cover each of the options in turn in the following sections.

3.1 Axiom

The option <axiom> allows us to take an axiom which we have defined in a trait, and add it to the list of hypotheses. The program below and its specification is intrinsically absurd, which demonstrates how we can manipulate the system to our own ends. We are only interested in the axioms and the VC in isolation.

```
--| TRAIT tee IS
--| INTRODUCES f: integer-> integer;
--| INTRODUCES g: integer-> integer;
--| AXIOMS:
--| ax1: (g(n)=(n*n));
--| ax2: ((x>=0)->(f(n)=(2*g(n))));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
PROCEDURE main(x : IN OUT integer)
  --| WHERE
  --|      IN (x>=0);
  --|      OUT (f(x)=((2*x)*x));
  --| END WHERE;
  --! VC Status: ** not proved **
  --! 1. (x>=0)
  --! >> (f(x)=((2*x)*x))
  --! <proof>

IS

BEGIN
  NULL;
END main;
```

Let us select <axiom> and add the parameters ax2 and tee, so that our VC now is as follows.

```
  --! VC Status: ** not proved **
  --! BY axiom ax2 in trait tee
  --! | 1. (x>=0)
  --! | 2. (FORALL x, n::((x>=0)->(f(n)=(2*g(n)))))
  --! | >> (f(x)=((2*x)*x))
  --! | <proof>
  --! □
```

We would need to bring in the other axiom at some stage to finish the proof. Notice that for convenience, all free variables in an axiom are assumed to be universally quantified, but this is shown explicitly when the axiom is added to the hypotheses.

3.2 Lemma

The <lemma> option is analogous to the <axiom> option. In a later version of Penelope, it will be required that lemmas are proved formally to follow from the axioms. At present they are regarded as known mathematical facts, and they have the same status as axioms.

3.3 Axm-subst

The <axm-subst> option works with an axiom which is expressed as an equality. It searches the goal for terms which match the term on the left of the equality sign in the axiom. It then substitutes the right hand side of the axiom for each occurrence it has found of the left. Suppose we begin with the following.

```
--| AXIOMS:
--| ax1: (g(n)=(n*n));
--| END AXIOMS;
  .
  .
  --! VC Status: ** not proved **
  --! 1. (x>=0)
  --! 2. (f(x)=(2*g(x)))
  --! >> (f(x)=(g(x)+(x*x)))
  --! <proof>
```

Then we may produce the following.

```
  --! VC Status: ** not proved **
  --! BY axiom ax1 in trait tee WITH n = x
```

```
--! | 1. (x>=0)
--! | 2. (f(x)=(2*(x*x)))
--! | >> (f(x)=((x*x)+(x*x)))
--! | <proof>
--! []
```

Observe that all of the free (universally quantified) variables in the axiom are instantiated by the user.

3.4 Lem-subst

The <lem-subst> option is analogous to <axm-subst>.

3.5 Axm-hyp

This option is like <axm-subst> in that it substitutes the right hand side of an axiom in the form of an equality, for the left hand side. However, rather than substituting for every occurrence in the goal, we select a particular hypothesis (by number), and a particular occurrence (by number from the left) within that hypothesis.

3.6 Lem-hyp

The <lem-hyp> option is analogous to <axm-hyp>.

3.7 Axm-concl

This option is like <axm-subst> in that it substitutes the right hand side of an axiom in the form of an equality, for the left hand side. However, rather than substituting for every occurrence in the goal, we select a particular occurrence (by number from the left) within the conclusion.

3.8 Lem-concl

The <lem-concl> option is analogous to <axm-concl>.

3.9 Arithmetic

The option <arithmetic> is a collection of frequently used facts about arithmetic operations which removes the need to reprove them continually. It includes things like `a+b=b+a` and `a*(b+c)=a*b+a*c` . The option <arithmetic> examines the conclusion of a goal (VC) and if it is a known true fact pronounces the goal "proved". The editor always attempts to use <arithmetic> automatically when it generates a VC during a proof, and so there is never any point in the user invoking it.

Here is an example program.

```
PROCEDURE test
  --| WHERE
  --|      GLOBAL a,b,c : OUT ;
  --|      OUT ((a*(b+c))=((a*b)+(a*c)));
  --| END WHERE;
  --! VC Status: proved
  --! BY arithmetic

IS

BEGIN
  NULL;
END test;
```

If we select show-VC from the previously hidden VC, then the VC is immediately proved by <arithmetic> and we do not get to see it.

3.10 Assume

If we wish to avoid doing a logical proof of a VC, but wish to have it regarded as nominally proved, we select the <assume> option, and make a comment to remind us of what we're up to. This will result in something like in the following example.

```
--! VC Status: proved
--! 1. (a>0)
--! >> ((a*b)>0)
--! BY assumption { skipping to get to something I want to investigate}
```

At first sight, this looks like cheating. However, it is not meant to be the case that the word "proved" on the screen stands as testimony to the fact that everything prior has been formally proved (although that could easily be enforced). If we desire that, we simply don't use <assume>. This feature is important for flexibility and experimentation. If there is one or more particular thing which we wish to investigate, we may choose to eliminate the clutter caused by a multitude of other unproved goals in which we have no interest. We use <assume> to do this; we may return to these other goals later and be a little more unassuming.

3.11 Case

To split up a proof into separate proofs according to different cases, which together exhaust all possibilities, we use the <case> option. For example, in

```
--! VC Status: ** not proved **
--! 1. (f(0)=1)
--! 2. (f(3)=1)
--! 3. ((x=0) OR (x=3))
--! >> (f(x)=1)
--! <proof>
```

we may wish to break the proof up according to the cases (x=0) and (x=3). The <case> option, however, does not assume that there is a convenient OR term as in the third member of the antecedent above. It also ensures that the various cases exhaust all possibilities. <case> adopts the sensible course and takes some boolean-valued <term> from the user, and then splits on <term> and (NOT <term>). Thus we obtain the following.

```
--! VC Status: ** not proved **
--! BY cases, using (x=0)
--! | 1. (f(0)=1)
--! | 2. (f(3)=1)
--! | 3. ((x=0) OR (x=3))
--! | 4. (x=0)
--! | >> (f(x)=1)
--! | <proof>
--! | 1. (f(0)=1)
--! | 2. (f(3)=1)
--! | 3. ((x=0) OR (x=3))
--! | 4. (NOT (x=0))
--! | >> (f(x)=1)
--! | <proof>
--! □
```

To break a proof up into more than two cases, we may use <case> again.

3.12 Contradict-goal

This proof-rule captures the familiar notion of proof by contradiction, where we assume the negation of what we are trying to prove, and then derive a contradiction. In the next release of Penelope it will be called "contradict-*conclusion*". The rule takes the conclusion, negates it and adds this to the hypotheses, and makes FALSE the new conclusion. If we begin with

```
--! VC Status: ** not proved **
--! 1. (NOT (b=0))
--! 2. (f(x)=(IF (x<4) THEN 0 ELSE b))
--! 3. (f(a)=1)
--! >> (a>=4)
--! <proof>
```

and use <contradict-goal>, we get the following.

```
--! VC Status: ** not proved **
--! BY contradiction
--! | 1. (NOT (b=0))
--! | 2. (f(x)=(IF (x<4) THEN 0 ELSE b))
--! | 3. (f(a)=1)
--! | 4. (NOT (a>=4))
--! | >> FALSE
--! | <proof>
--! []
```

3.13 Contradict-hypothesis

This rule is a variant on the notion that from inconsistent hypotheses one can prove anything. It works on the fact that a set of statements is inconsistent if and only if the negation of any single one of them can be proved from the remaining statements. We disregard the old conclusion, remove one of the hypotheses, and make the new conclusion the negation of this statement. Thus, from

```
--! VC Status: ** not proved **
--! 1. (x>=0)
--! 2. (x<0)
--! 3. ((x>=0)->(f(x)=5))
--! >> (f(x)=f((2*x)))
```

we may obtain the following.

```
--! VC Status: ** not proved **
--! BY contradiction, in 2
--! | 1. (x>=0)
--! | 2. ((x>=0)->(f(x)=5))
--! | >> (NOT (x<0))
--! | <proof>
--! □
```

3.14 Cut

This is a very important rule. Sometimes we see that the best way to get from the hypotheses to the conclusion is through a certain intermediate statement φ. More precisely, we see that we can prove φ from the hypotheses, and that adding φ to the hypotheses will enable us to prove the conclusion. In this case we use the <cut> option and this gives us those two tasks as subgoals. In ordinary language, we say that we "cut on φ".

Suppose that we start with the following.

```
--! VC Status: ** not proved **
--! 1. ((x-5)>0)
--! 2. ((x+5)<15)
--! 3. (((x>5) AND (x<10))->(f(x)=0))
--! >> (f(x)=0)
```

We observe that from 1 and 2 we can prove that `5<x<10` and that from this and 3 we can obtain the desired conclusion. So we decide to cut on `5<x<10`, or, in the assertion language, `(x>5 AND x<10)`.

```
--! VC Status: ** not proved **
--! BY cut with ((x>5) AND (x<10))
--! | 1. ((x-5)>0)
--! | 2. ((x+5)<15)
--! | 3. (((x>5) AND (x<10))->(f(x)=0))
--! | >> ((x>5) AND (x<10))
--! | <proof>
--! []
--! | 1. ((x-5)>0)
--! | 2. ((x+5)<15)
--! | 3. (((x>5) AND (x<10))->(f(x)=0))
--! | 4. (x>5)
--! | 5. (x<10)
--! | >> (f(x)=0)
--! | <proof>
--! []
```

Each of these subgoals must now be tackled separately. Notice that in the second one, the statement which we cut on has been split up into two parts by the automatic simplifier.

3.15 Equals-synthesis

The reader may have noticed that there are many rules called "synthesis" rules, and for each one, a parallel "analysis" rule. We begin this section with a word about synthetic rules in general (analytic rules will be described when we reach the first example). Synthesis rules are those which we use to build, or "synthesize", the conclusion of a goal sequent, from subgoals where it does not appear. Remember that we are using rules backwards, and so we start with a goal with a structured conclusion and ask "How can we construct this from subgoals?".

This option can be used if the conclusion of our goal is an equality, and currently is to be used only if the equality is between two boolean valued terms. The option <equals-synthesis> generates two subgoals, each of which requires us to prove one side of the equality from the original hypotheses plus the other side. Suppose that we begin with the following.

```
--! VC Status: ** not proved **
--! 1. a
--! 2. b
--! >> ((a->b)=(a OR b))
--! <proof>
```

Selecting <equals-synthesis> we obtain two subgoals as in the following.

```
--! VC Status: ** not proved **
--! BY synthesis of =
--! | 1. a
--! | 2. b
--! | 3. (a->b)
--! | >> (a OR b)
--! | <proof>
--! □
--! | 1. a
--! | 2. b
--! | 3. (a OR b)
--! | >> (a->b)
--! | <proof>
--! □
```

3.16 expand-array-elts

When an assertion-language statement about an array undergoes predicate transformation, the information about what an Ada statement does to the array is represented in a special form. To see this, observe the following chain of preconditions, beginning with the last and working back.

```
BEGIN
  --: PRECONDITION = (((((a[i=>val1])[j=>val2])[k=>val3])[n])=2);
  a(i):=val1;
  --: PRECONDITION = ((((a[j=>val2])[k=>val3])[n])=2);
  a(j):=val2;
  --: PRECONDITION = (((a[k=>val3])[n])=2);
  a(k):=val3;
  --: PRECONDITION = ((a[n])=2);
END main;
```

As is fairly obvious, a[k=>val3] means the array a with the k*th* entry replaced by val3.

Now let us look at the whole subprogram with the above body.

```
.
.
  TYPE intarray IS ARRAY(integer) OF integer;
  a : intarray;
  PROCEDURE test
    --| WHERE
    --|       GLOBAL a,n : OUT ;
    --|       OUT ((a[n])=2);
    --| END WHERE;
    --! VC Status: ** not proved **
    --! >> (((((a[i=>val1])[j=>val2])[k=>val3])[n])=2)
    --! <proof>

  IS
    i, j, k, val1, val2, val3 : integer;

  BEGIN
    a(i):=val1;
    a(j):=val2;
    a(k):=val3;
  END test;
.
.
```

We need to expand the array representation in the VC so that we have the VC expressed in the more basic and familiar "predicate calculus core" of our assertion language. This expansion can in general be a time-consuming task, and so we do not have it done automatically. To do it the user selects <expand-array-elts>. In our example, the VC goes from

```
--! VC Status: ** not proved **
--! >> (((((a[i=>val1])[j=>val2])[k=>val3])[n])=2)
--! <proof>
```

to

```
--! VC Status: ** not proved **
--! BY array simplification
--! | >> (IF (k=n) THEN (val3=2)
         ELSE (IF (j=n) THEN (val2=2)
               ELSE (IF (i=n) THEN (val1=2) ELSE ((a[n])=2))))
--! | <proof>
--! □
```

3.17 Or-syn-left

This rule has an analogous sister rule <or-syn-right>. To prove an OR statement, it is sufficient to prove one of the disjuncts. If we think that we can prove the left-hand disjunct, we use <or-syn-left> to give us the single subgoal which has this as its conclusion, and the same hypotheses. Thus if we begin with

```
--! VC Status: ** not proved **
--! 1. (a=(b-1))
--! 2. (b=2)
--! >> ((a=1) OR (b=2))
--! <proof>
```

we should select <or-syn-left> to get the following.

```
--! BY left synthesis of OR
--! | 1. (a=(b-1))
--! | 2. (b=2)
--! | >> (a=1)
--! | <proof>
--! []
```

Of course there are OR conclusions for which it is impossible to prove one of the disjuncts by itself. In a case like

```
--! VC Status: ** not proved **
--! 1. (1<x)
--! 2. (x<4)
--! >> ((x=2) OR (x=3))
--! <proof>
```

we would not use <or-syn-left>, as this would leave us with the unprovable goal

```
--! VC Status: ** not proved **
--! BY left synthesis of OR
--! | 1. (1<x)
--! | 2. (x<4)
--! | >> (x=2)
--! | <proof>
--! []
```

.

3.18 Or-syn-right

This rule is entirely analogous to the previous <or-syn-left> — it takes the right-hand disjunct of an OR statement as the conclusion of the generated subgoal.

3.19 Simplify

This option makes use of a rather *ad hoc* collection of rules designed to simplify the goal. It is always applied automatically before a goal is produced, to replace prolixities like φ AND φ and ψ->TRUE by φ and ψ respectively. As it is applied automatically, there is seldom any need for the user to invoke it. Multiple applications of <simplify> will sometimes do more than one, and thus occasionally this rule is useful.

3.20 SDVS-simplify

The SDVS simplifier is a powerful component which contains a decision procedure for a useful fragment of the logic of the assertion language. The simplifier is software code produced by the Aerospace Corporation, which implements the Nelson-Oppen "co-operating decision procedure" (see [6]). In the event that the SDVS simplifier is unable to simplify a goal to TRUE, it returns the best simplification that it can make. For the details of how the SDVS simplifier works and what it does, see [7]. We typically select <SDVS-simplify> after the previous proof-rule has left us with a large, messy-looking goal, and then we begin analyzing the cleaned-up version. The great virtue of this option is that it frees us from the chore of dealing piece by piece with a mass of trivial detail. It essentially "factors out" the trivia and enables us to deal with the major structural directions in which the proof has to go.

3.21 Prenex-simplify

This is an experimental rule and we will not deal with it.

3.22 Subst-l

This rule has a sister rule <subst-r>. Suppose that we have an equality as one of the hypotheses of our goal. The terms on either side of the equality may occur in other places throughout the sequent, of course. Suppose that we wish to substitute, uniformly, the term on the right of the equality for the term on the left. Then we use <subst-l>. For example, we may begin with

```
--! VC Status: ** not proved **
--! 1. (y>2)
--! 2. (FORALL z::((z>2)->(f(z)=0)))
--! 3. (x=y)
--! >> (f(x)=0)
--! <proof>
```

and progress to the following.

```
--! VC Status: ** not proved **
--! BY left substitution of 3
--! | 1. (y>2)
--! | 2. (FORALL z::((z>2)->(f(z)=0)))
--! | >> (f(y)=0)
--! | <proof>
--! □
```

This substitution works not only for simple variables as in the example, but for arbitrarily complicated terms.

This rule is our first to involve substitution. Naturally, substitutions are done only for occurrences of terms in which all variables are free. When a substitution is done, collisions are avoided by renaming bound variables. The following example illustrates these remarks simply, and is not a valid sequent. From

```
--! VC Status: ** not proved **
--! 1. (FORALL u::((x+y)=u))
--! 2. (FORALL y::((x+y)=k))
--! 3. ((x+y)=(u+v))
--! >> a
--! <proof>
```

we may get

```
--! VC Status: ** not proved **
--! BY left substitution of 3
--! | 1. (FORALL u1::((u+v)=u1))
--! | 2. (FORALL y::((x+y)=k))
--! | >> a
--! | <proof>
--! □
```

3.23 Subst-r

This rule is entirely analogous to the previous rule, <subst-l>. This option, <subst-r>, takes the left-hand side of an equality and substitutes it for occurrences of the right-hand side.

3.24 Thinning

During the course of a proof, it often happens that the list of hypotheses of the current goal contains some hypotheses which are of no use— intuitively, they are not needed to ensure that the conclusion follows from the hypotheses. Unless the goal is very simple, it is desirable to remove these unwanted hypotheses for two reasons. One is that the removal of unnecessary clutter simplifies the presentation and helps us to analyze the situation. The other is that when the SDVS simplifier is invoked, it works with the whole sequent, and so unnecessary hypotheses slow the system down needlessly.

If we begin with

```
--! VC Status: ** not proved **
--! 1. (x>0)
--! 2. (x<5)
--! 3. (FORALL y::((y>0)->(f(y)=1)))
--! 4. (x<7)
--! >> (f(x)=1)
--! <proof
```

then we can remove the unnecessary hypotheses 2 and 4 to generate the following subgoal.

```
--! VC Status: ** not proved **
--! BY thinning 2, 4
--! | 1. (x>0)
--! | 2. (FORALL y::((y>0)->(f(y)=1)))
--! | >> (f(x)=1)
--! | <proof>
--! □
```

3.25 And-anal

As described above, a synthesis rule takes a goal and builds the conclusion out of its component parts. An "analysis" rule takes a goal and pulls one of the hypotheses apart to give us access to the components, so that we have more raw material from which to build our conclusion. The rule <and-anal> takes a goal with a conjunction (φ AND ψ) as an hypothesis, and produces a subgoal with φ and ψ as separate hypotheses. This rule is always applied automatically, and so there will not be any need for the user to apply it manually. However, the user must understand what it does, and so we will describe it.

If we began with the goal

```
1. ((0<x) AND (x<5))
2. (FORALL x::((x>0)->(f(x)=1)))
>> (f(x)=1)
```

and applied <and-anal> then we would get the following.

```
1. (0<x)
2. (x<5)
3. (FORALL x::((x>0)->(f(x)=1)))
>> (f(x)=1)
```

We repeat that the above scenario is impossible to produce on the screen, as the first sequent would never appear, being automatically reduced to the second.

3.26 And-syn

To prove an AND statement, it is sufficient to prove each of the conjuncts separately. If we begin with

```
--! VC Status: ** not proved **
--! 1. (5<x)
--! 2. (x<10)
--! >> ((0<x) AND (x<15))
--! <proof>
```

and apply <and-syn> we obtain the following.

```
--! VC Status: ** not proved **
--! BY synthesis of AND
--! | 1. (5<x)
--! | 2. (x<10)
--! | >> (0<x)
--! | <proof>
--! | 1. (5<x)
--! | 2. (x<10)
--! | >> (x<15)
--! | <proof>
--! □
```

3.27 Conflicting-hypotheses

This rule is the familiar "from a contradiction you can prove anything". If the goal has an hypothesis φ, and another hypothesis (NOT φ), the rule <conflicting-hypotheses> recognizes it as an axiom and the goal is pronounced proved. This rule is always applied automatically, so there will not be any need for the user to apply it manually. It is important to understand this rule, so that the user knows that is desirable to drive the proof to the point where it is applied, if that is possible.

If we began with the goal

```
    1. (x=2);
    2. (NOT (x=2));
    >> (y=0);
```

then the application of <conflicting-hypotheses> would result in the goal being pronounced proved (it is an axiom of the sequent calculus). Of course, after applying any rule which officially produces a subgoal like the above, we would not get to see the subgoal as it would be dealt with immediately.

3.28 Exists-anal

If our goal has an existential statement as one of the hypotheses, we typically wish to instantiate it. It is then that we use <exists-anal>. If we begin with

```
--! VC Status: ** not proved **
--! 1. (EXISTS x::((a<x) AND (x<b)))
--! >> (a<b)
--! <proof
```

and instantiate, we get the following.

```
--! VC Status: ** not proved **
--! BY analysis of EXISTS, in 1
--! | 1. (EXISTS x::((a<x) AND (x<b)))
--! | 2. (a<x)
--! | 3. (x<b)
--! | >> (a<b)
--! | <proof>
--! □
```

Notice that the system has chosen x as the free variable of instantiation. It has chosen the variable which is bound in hypothesis 1, to remind the user where it came from, though of course there is no *logical relationship* between the two occurrences. There are some restrictions on the choice of variable. From the original existential statement, we can infer nothing about the relationship between the variable on which we instantiate, and the other hypotheses and conclusion. Thus the variable chosen must not occur free in any hypothesis or the conclusion. So, from

```
--! VC Status: ** not proved **
--! 1. (x>0)
--! 2. (EXISTS x::((a<x) AND (x<b)))
--! >> (a<b)
--! <proof>
```

we get the following.

```
--! VC Status: ** not proved **
--! BY analysis of EXISTS, in 2
--! | 1. (x>0)
--! | 2. (EXISTS x::((a<x) AND (x<b)))
--! | 3. (a<x1)
--! | 4. (x1<b)
--! | >> (a<b)
--! | <proof>
--! □
```

Note here how <and-anal> has been applied automatically.

3.29 Exists-syn

To prove a goal sequent with an existential conclusion, it is sufficient to prove a subgoal where the conclusion has a particular value in place of the existentially quantified variable. For example, to establish

```
--! VC Status: ** not proved **
--! 1. (a<(u+v))
--! 2. ((u+v)<b)
--! >> (EXISTS x::((a<x) AND (x<b)))
--! <proof
```

we use <vexists-syn> to produce the following subgoal, where we have chosen the term (u+v) to replace the variable **x**.

```
--! VC Status: ** not proved **
--! BY synthesis of EXISTS
--! exhibiting (u+v)
--! | 1. (a<(u+v))
--! | 2. ((u+v)<b)
--! | >> ((a<(u+v)) AND ((u+v)<b))
--! | <proof>
--! []
```

The system must avoid collisions of variables. If we begin with

```
--! VC Status: ** not proved **
--! 1. (a<(u+(x+y)))
--! 2. ((u+(x+y))<b)
--! >> (EXISTS x, y::(((a<x) AND (x<b)) AND (x<y)))
--! <proof>
```

and wish to produce a subgoal by putting (u+x+y) in place of **x** in the conclusion, we obtain the following.

```
--! VC Status: ** not proved **
--! BY synthesis of EXISTS
--! exhibiting ((u+x)+y)
--! | 1. (a<(u+(x+y)))
--! | 2. ((u+(x+y))<b)
--! | >> (EXISTS y1::(((a<((u+x)+y)) AND (((u+x)+y)<b)) AND (((u+x)+y)<y1)
--! | <proof>
--! □
```

We see that the x was no problem, but the system re-named the quantified variable to avoid capturing the y.

3.30 Forall-anal

If we have a universal statement as an hypothesis , we can instantiate it on any term, and add the result to the hypotheses, by using <forall-anal>. For example, beginning with

```
--! VC Status: ** not proved **
--! 1. (y=3)
--! 2. (FORALL x::((x>0)->(f(x)=1)))
--! >> (f(y)=1)
--! <proof>
```

we may may generate the following subgoal.

```
--! VC Status: ** not proved **
--! BY analysis of FORALL, in 2 with y
--! | 1. (y=3)
--! | 2. (FORALL x::((x>0)->(f(x)=1)))
--! | 3. ((y>0)->(f(y)=1))
--! | >> (f(y)=1)
--! | <proof>
--! □
```

Sometimes, bound variables are re-named to avoid collisions.

3.31 Forall-syn

In order to establish a goal with a universal conclusion, it suffices to prove the subgoal where the conclusion has the universal quantifier removed, and an appropriate "dummy" in place of the previously quantified variable. Starting with

```
--! VC Status: ** not proved **
--! 1. (FORALL y::((y>0)->(f(y)=2)))
--! >> (FORALL x::((x>7)->(f(x)=2)))
--! <proof>
```

we select <forall-syn> to get the following.

```
--! VC Status: ** not proved **
--! BY synthesis of FORALL
--! | 1. (FORALL y::((y>0)->(f(y)=2)))
--! | >> ((x>7)->(f(x)=2))
--! | <proof>
--! □
```

From here, we would instantiate hypothesis 1 using x, to proceed with the proof. The system chooses the variable for us, ensuring that it does not occur free anywhere else.

3.32 False-anal

Intuitively, this rule is related to <conflicting-hypotheses>. It says that if we have FALSE explicitly as an hypothesis, then the goal is pronounced proved. All such sequents are in fact axioms of our calculus. This rule is always applied automatically, and so the user will never need to apply it manually. Examples of its use do not abound, but we will conjure one up for the occasion.

Suppose we have the goal

```
--! VC Status: ** not proved **
--! 1. (a=b)
--! 2. (b=c)
--! 3. (NOT (a=c))
--! >> (x<y)
--! <proof>
```

and observe that the hypotheses are contradictory, though not explicitly so. We may wish to cut on FALSE, which would give the following.

```
--! VC Status: ** not proved **
--! BY cut with FALSE
--! | 1. (a=b)
--! | 2. (b=c)
--! | 3. (NOT (a=c))
--! | >> FALSE
--! | <proof>
--! □
--! | BY analysis of FALSE
--! □
```

One of the subgoals produced by the cut was

```
1. (a=b)
2. (b=c)
3. (NOT (a=c))
4. FALSE
>> (x<y)
```

but we never got to see it because it was automatically recognized and pronounced proved by <false-anal>. Note that cutting on FALSE above was not really the most sensible way to go.

3.33 False-syn

This rule says that if we have a goal with explicitly contradictory hypotheses, and **FALSE** as the conclusion, it is pronounced proved as an axiom. It is a special case of <conflicting-hypotheses>, and is never really needed. Indeed, <conflicting-hypotheses> is always applied automatically, and so the user will never need to apply <false-anal> manually, *a fortiori*.

For the sake of completeness, <false-anal> would take

```
1. (a=b)
2. (NOT (a=b))
>>  FALSE
```

and pronounce it proved, if it ever got the chance.

3.34 If-pair

Suppose that we have an **if-then-else** statement as an hypothesis of a goal. This can be used, if the conclusion is also an **if-then-else** statement, without having to show that the boolean condition, or its negation, follows from the other hypotheses. The rule for this, <if-pair>, is best understood by looking at a highly abstract example, which is rather like a formal proof rule.

```
--! VC Status: ** not proved **
--! 1. hypothesis1
--! 2. (IF e THEN a ELSE b)
--! >> (IF e THEN a_prime ELSE b_prime)
--! <proof>
```

With this as our goal, if we select <if-pair>, we obtain the following two subgoals.

```
--! VC Status: ** not proved **
--! BY pairing of IF
--! | 1. hypothesis1
--! | 2. e
--! | 3. a
--! | >> a_prime
--! | <proof>
--! □
--! | 1. hypothesis1
--! | 2. (NOT e)
--! | 3. b
--! | >> b_prime
--! | <proof>
--! □
```

3.35 If-syn

To prove a goal which has an `if-then-else` statement as its conclusion, we use <if-syn>. It slits the goal up into two cases, in an intuitively clear fashion.

If we begin with

```
--! VC Status: ** not proved **
--! 1. (FORALL x::((x<5)->(f(x)=0)))
--! 2. (FORALL x::((x>2)->(g(x)=1)))
--! >> (IF (x<3) THEN (f(x)=0) ELSE (g(x)=1))
--! <proof>
```

then the selection of <if-syn> gives us the following.

```
--! VC Status: ** not proved **
--! BY synthesis of IF,
--! | 1. (FORALL x::((x<5)->(f(x)=0)))
--! | 2. (FORALL x::((x>2)->(g(x)=1)))
--! | 3. (x<3)
--! | >> (f(x)=0)
--! | <proof>
--! []
--! | 1. (FORALL x::((x<5)->(f(x)=0)))
--! | 2. (FORALL x::((x>2)->(g(x)=1)))
--! | 3. (NOT (x<3))
--! | >> (g(x)=1)
--! | <proof>
--! []
```

3.36 If-then-anal

This rule enables us to make use of an `if-then-else` statement in the hypotheses, in the case where the boolean condition follows from the other hypotheses. For example, suppose we have

```
--! VC Status: ** not proved **
--! 1. (a=b)
--! 2. (b=c)
--! 3. (IF (a=c) THEN (x=0) ELSE (x=1))
--! >> ((x+x)=x)
--! <proof>
```

We note that the boolean condition is true in the light of hypotheses 1 and 2, and would like to enlist the help of the "THEN" part in establishing the conclusion. We use <if-then-anal> to produce the following.

```
--! VC Status: ** not proved **
--! BY then analysis of IF, in 3
--! | 1. (a=b)
--! | 2. (b=c)
--! | >> (a=c)
--! | <proof>
--! □
--! | 1. (a=b)
--! | 2. (b=c)
--! | 3. (a=c)
--! | 4. (x=0)
--! | >> ((x+x)=x)
--! | <proof>
--! □
```

There are two subgoals—one is to show that the boolean condition is indeed a consequence of the other hypotheses, and the other is to use the "THEN" part as described above.

3.37 If-else-anal

This rule is similar to the previous rule, <if-then-anal>. In this case, we show that the boolean condition is false, and use the "ELSE" clause to help prove our conclusion. For example, from

```
--! VC Status: ** not proved **
--! 1. (a=(b+1))
--! 2. (IF (a=b) THEN (x=0) ELSE (x=1))
--! >> ((x+x)=(2*x))
--! <proof>
```

we get the following.

```
--! VC Status: ** not proved **
--! BY else analysis of IF, in 2
--! | 1. (a=(b+1))
--! | >> (NOT (a=b))
--! | <proof>
--! □
--! | 1. (a=(b+1))
--! | 2. (NOT (a=b))
--! | 3. (x=1)
--! | >> ((x+x)=(2*x))
--! | <proof>
--! []
```

3.38 Imp-anal

The idea behind <imp-anal> is that if we have an implication φ->ψ in the hypotheses of a goal, then provided we can establish φ, we can add ψ (and φ) to the hypotheses.

Suppose that we have

```
--! VC Status: ** not proved **
--! 1. (x=3)
--! 2. ((x>0)->(f(x)=(x+1)))
--! >> (f(x)=4)
--! <proof>
```

as our goal. Then, selecting <imp-anal>, we obtain the following subgoal.

```
--! VC Status: ** not proved **
--! BY analysis of IMPLIES, in 2
--! | 1. (x=3)
--! | >> (x>0)
--! | <proof>
--! []
--! | 1. (x=3)
--! | 2. (f(x)=(x+1))
--! | 3. (x>0)
--! | >> (f(x)=4)
--! | <proof>
--! []
```

3.39 Imp-syn

If the conclusion of our goal is an implication, and we want to synthesize it from its component parts, we use <imp-syn>. This rule is always applied automatically, and so the user will never have to apply it manually. A goal such as

```
1. (FORALL x::((x>0)->(f(x)=g(x))))
2. (g(x)=1)
>> ((x=3)->(f(x)=1))
```

would be automatically be reduced by <imp-syn> to

```
1. (FORALL x::((x>0)->(f(x)=g(x))))
2. (g(x)=1)
3. (x=3)
>> (f(x)=1)
```

.

3.40 Hypoth

If we have a goal sequent in which the conclusion also appears as one of the hypotheses, the rule <hypothesis> will pronounce it proved. All such sequents are in fact axioms of our logical system. The rule is always applied automatically, and so the user will never need to apply it manually. Naturally, it is useful to aim for a point where it is applied (automatically) when directing a proof, so we must understand its function.

If the result of a previous rule was officially the goal

```
1.  (FORALL x::((x>0)->(f(x)=1)));
2.  (y=3);
3.  (f(y)=1);
4.  (y>0);
>>  (f(y)=1);
```

then it would be pronounced proved by <hypothesis>, without us getting to see it.

3.41 Not-anal

This rule says that if our goal has (NOT φ) as an hypothesis, then it suffices that φ should follow from the other hypotheses. It is really a version of the <contradict-hypothesis> rule discussed above, but in a form which is more convenient for when the hypothesis in question is a NOT statement—we do not produce a double negation in the conclusion of the subgoal.

Suppose we have

```
--! VC Status: ** not proved **
--! 1. (NOT (x>0))
--! 2. (x=3)
--! >> (f(x)=1)
--! <proof>
```

and apply <not-anal>. Then we have the following.

```
--! VC Status: ** not proved **
--! BY contradiction, in 1
--! | 1. (x=3)
--! | >> (x>0)
--! | <proof>
--! []
```

In the Penelope extract here, we see the results attributed to "`contradiction`". This is because <not-anal> is implemented as a combination of other rules, including <contradict-hypothesis>. Also use of <contradict-hypothesis> will give us the same result as we obtain with <not-anal>, as the system automatically deals with double negations.

3.42 Not-syn

This rule is a version of proof by contradiction. If our goal has (NOT φ) as the conclusion, then it suffices that if φ is added to the hypotheses, FALSE follows as a conclusion. It is in essence a variant of the <contradict-goal> rule discussed above, but in a form which is more convenient for when the conclusion is a NOT statement—we do not produce a double negation in new hypothesis added to the subgoal.

If we begin with

```
--! VC Status: ** not proved **
--! 1. (FORALL x::((x>0)->(f(x)=1)))
--! 2. (x=3)
--! >> (NOT (f(x)=2))
--! <proof>
```

then <not-syn> gives us

```
--! VC Status: ** not proved **
--! BY contradiction
--! | 1. (FORALL x::((x>0)->(f(x)=1)))
--! | 2. (x=3)
--! | 3. (f(x)=2)
--! | >> FALSE
--! | <proof>
--! □
```

In fact, <contradict-goal> will give us the same result, as the system automatically deals with double negations.

3.43 Or-anal

If we have a goal with an OR statement as one of the hypotheses, then we break the proof up into two cases, showing that the conclusion follows when either of the disjuncts is true. If our goal is

```
--! VC Status: ** not proved **
--! 1. (FORALL x::((x>0)->(f(x)=1)))
--! 2. ((x=3) OR (x=5))
--! >> (f(x)=1)
--! <proof>
```

then the selection of <or-anal> gives us the following two subgoals.

```
--! VC Status: ** not proved **
--! BY analysis of OR, in 2
--! | 1. (FORALL x::((x>0)->(f(x)=1)))
--! | 2. (x=3)
--! | >> (f(x)=1)
--! | <proof>
--! | 1. (FORALL x::((x>0)->(f(x)=1)))
--! | 2. (x=5)
--! | >> (f(x)=1)
--! | <proof>
--! []
```

3.44 True-syn

If we had a goal which had TRUE as the conclusion, <true-syn> would pronounce it proved. In fact, all such sequents are axioms of our logical system. The rule is applied automatically, and so there is never any need for the user to apply it manually.

If we did produce a goal

```
1.  (FORALL x::((x>0)->(f(x)=1)));
2.  (y=3);
3.  (f(y)=2);
>>  TRUE
```

then it would be pronounced proved without the goal being displayed. Examples where this arises are rare. It is conceivable that someone may write a program with TRUE as the postcondition (by specifying TRUE as the OUT condition, or, equivalently, having no OUT condition). In this case, the <true-syn> rule completes the proof.

When using the prover, many proofs terminate successfully with the announcement that the proof was completed "`BY synthesis of TRUE`", though the user did not invoke <true-syn> . The reason is that many of the automatic simplification routines attempt to reduce the goal to a "normal form" with TRUE as the conclusion. If one is successful, the result is handed to <true-syn> which produces the announcement in question.

Chapter 4

Loops

We now begin to study our first non-trivial examples. In this chapter we are concerned with loops, which are one of the significant causes of complexity in program verification. In the current version of Penelope, we deal with the **while** loop, and the **simple** loop without an iteration scheme. The **for** loop is not dealt with now, but will be in future versions.

The first loop we will look at is the traditional **while** loop. When we select <while-loop>, we get the following template.

```
--! VC Status: hidden
--! []
WHILE <exp> LOOP
  --| INVARIANT  <term>;
  <statement>
END LOOP;
```

We will show the logical structure of verification involving a **while** loop in Penelope. Suppose we enter Ada code to produce the desired loop

```
--! VC Status: hidden
--! []
WHILE b LOOP
  --| INVARIANT  <term>;          ...(1)
  loop_body;
END LOOP;
```

in a program we wish to verify.

We must add an invariant, which we will call **inv**, on line (1). We choose **inv** so that if the program begins execution in any allowed (by the specification) state, then inv will be true immediately before **loop_body** is executed for the first time. Furthermore, inv must remain true after each subsequent execution. Our inv must be powerful enough that if b is false (on leaving the loop), then inv and (NOT b) together ensure that the postcondition holds. The loop precondition is inv. In order to construct the loop VC, Penelope transforms inv up from the bottom of loop_body, to produce what we will call inv', which is roughly the weakest precondition. Schematically, the logic of the loop is as follows.

```
.
.
--: PRECONDITION = inv;
--! VC Status: ** not proved **
--! 1. inv
--! >> (IF b THEN inv' ELSE postcond)
--! <proof>
WHILE b LOOP
  --|INVARIANT = inv;
  loop_body;
END LOOP;
--: PRECONDITION = postcond;
.
.
```

Note that this is not actually a possible Penelope display—for example, in place of our invention inv' Penelope would put the actual result of transforming inv.

Now we will look at some real examples. Our first example will be done in some detail. This will be useful to the reader who is unfamiliar with Penelope, but is also designed to be useful to the reader who is not very experienced with verification in general. In this latter context we make various background remarks, give some heuristics for constructing invariants and make a couple of typical wrong turns from which we then recover. Later examples will become more streamlined.

4.1 Maximum element of an array

We wish to define a function which takes an array and returns its maximum element. Let us for simplicity define a function arraymax which takes two arguments: an array a and an integer n. The function returns the maximum value for an index between 1 and n. Suppose we enter just the Ada code and get the following.

```
--> 3 VCs NOT SHOWN!
PROCEDURE proc
  --| WHERE
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS
  TYPE arrayn IS ARRAY(integer) OF integer;
  FUNCTION arraymaxof(a : IN arrayn; n : IN integer) RETURN integer
    --| WHERE
    --| END WHERE;
    --! VC Status: hidden
    --! []

  IS
    m : integer := a(1);
    i : integer := 2;

  BEGIN
    --! VC Status: hidden
    --! []
    WHILE (i<=n) LOOP
      --| INVARIANT  <term>;
      IF (m<a(i)) THEN
        m:=a(i);
      END IF;
      i:=(i+1);
    END LOOP;
    RETURN  m;
  END arraymaxof;
BEGIN
  NULL;
END proc;
```

Let us now give the specifications for the function. One way would be to give a definition in the specification part, of the value returned by the Ada function **arraymax**, as follows.

```
  FUNCTION arraymaxof(a : IN arrayn; n : IN integer) RETURN integer
    --> GLOBAL ();
    --| WHERE
    --|      RETURN x SUCH THAT
                      (FORALL j::(((1<=j) AND (j<=n))->((a[j])<=x)));
    --|      RETURN x SUCH THAT
                      (EXISTS k::(((1<=k) AND (k<=n)) AND ((a[k])=x)));
    --| END WHERE;
```

The VCs then are what we need to prove to establish that the Ada function **arraymax** really returns values with these properties. However, we will give a cleaner presentation and illustrate the use of traits at the same time by using an alternative method.

Another way to specify the program above is to state that the value returned by the Ada function **arraymax** for a given input is the value returned by an abstract mathematical function **matarrmax** on the same input. Such a mathematical function has the same status as ordinary mathematical functions like plus, times, cube root, standard deviation, etc.. This new function must be defined inductively, and handed to the logical system in a module called a *trait*:

```
--| TRAIT t IS
--| INTRODUCES matarrmax: intarray, integer-> integer;
--| AXIOMS:
--| bc: (matarrmax(ar, 1)=(ar[1]));
--| istep: (matarrmax(ar, (k+1))=(IF (matarrmax(ar, k)>(ar[(k+1)]))
                                  THEN matarrmax(ar, k)
                                  ELSE (ar[(k+1)])));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
```

Following INTRODUCES, we give the signature of the function (in general we can have several INTRODUCES clauses if we need several functions). Following AXIOMS, we state various properties of the function. Here, we have stated enough to define the function completely for positive integers, by giving the obvious inductive definition.

In an axiom, all free variables, such as ar and k here, are assumed to be universally quantified. An array component given as a(k) in Ada corresponds to a[k] in the assertion language. The square brackets are used to facilitate overload resolution, since they immediately distinguish the occurrence of a from one which refers to an homonymous function a.

There is no need to have just a single mathematical function which is used in the verification of the Ada function as we have here. We can define any number of functions and use them in any fashion we choose to specify the Ada function or Ada program. In fact it is highly desirable to build a library of mathematical functions and use them in traits whenever they are useful for a specification.

Here then is our fully specified program.

```
--> 3 VCs NOT SHOWN!
--| TRAIT t IS
--| INTRODUCES matarrmax: intarray, integer-> integer;
--| AXIOMS:
--| bc: (matarrmax(ar, 1)=(ar[1]));
--| istep: (matarrmax(ar, (k+1))=(IF (matarrmax(ar, k)>(ar[(k+1)]))
                                  THEN matarrmax(ar, k)
                                  ELSE (ar[(k+1)])));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
PROCEDURE proc
  --| WHERE
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  TYPE arrayn IS ARRAY(integer) OF integer;
  FUNCTION arraymaxof(a : IN arrayn; n : IN integer) RETURN integer
    --| WHERE
    --|      IN (n>=1);
    --|      RETURN x SUCH THAT (x=matarrmax(a, n));
    --| END WHERE;
    --! VC Status: hidden
    --! []

  IS
    m : integer := a(1);
    i : integer := 2;

  BEGIN
    --! VC Status: hidden
    --! []
    WHILE (i<=n) LOOP
      --| INVARIANT  <term>;
      IF (m<a(i)) THEN
        m:=a(i);
      END IF;
      i:=(i+1);
    END LOOP;
    RETURN  m;
  END arraymaxof;
BEGIN
  NULL;
END proc;
```

We must now choose our invariant. This is really the most challenging part of the verifier's task. Gries's book [2] contains heuristics for choosing invariants, and the reader should look there if he needs more background. At a typical intermediate stage, an execution of our loop body gives, in m, the maximum value for indices 1 to i-1, and increases i each time through until it reaches n+1. Thus our first, and most important, contribution to the invariant is the statement that m is the maximum array value for indices 1 to i-1. Let us add this and show the loop and program VCs which result.

```
--| TRAIT t IS
--| INTRODUCES matarrmax: intarray, integer-> integer;
--| AXIOMS:
--| bc: (matarrmax(ar, 1)=(ar[1]));
--| istep: (matarrmax(ar, (k+1))=(IF (matarrmax(ar, k)>(ar[(k+1)]))
                                  THEN matarrmax(ar, k)
                                  ELSE (ar[(k+1)])));
--| END AXIOMS;
.
.
  TYPE arrayn IS ARRAY(integer) OF integer;
  FUNCTION arraymaxof(a : IN arrayn; n : IN integer) RETURN integer
    --| WHERE
    --|      IN (n>=1);
    --|      RETURN x SUCH THAT (x=matarrmax(a, n));
    --| END WHERE;
    --! VC Status: ** not proved **
    --! 1. (n>=1)
    --! >> ((a[1])=matarrmax(a, 1))
    --! <proof>

  IS
    m : integer := a(1);
    i : integer := 2;

  BEGIN
    --! VC Status: ** not proved **
    --! 1. (m=matarrmax(a, (i-1)))
    --! >> (IF (i<=n)
              THEN (IF (m<(a[i]))
                    THEN ((a[i])=matarrmax(a, i))
                    ELSE (m=matarrmax(a, i)))
              ELSE (m=matarrmax(a, n)))
    --! <proof>
    WHILE (i<=n) LOOP
      --| INVARIANT  (m=matarrmax(a, (i-1)));
      IF (m<a(i)) THEN
        m:=a(i);
      END IF;
      i:=(i+1);
    END LOOP;
    RETURN  m;
  END arraymaxof;
.
.
```

We will deal with the loop VC first. The first thing that springs to mind is that the main "THEN" clause requires the inductive step of the definition of `matarrmax` in the trait. We bring that down into the hypotheses with the <axiom> component of the proof editor.

```
    --! VC Status: ** not proved **
    --! BY axiom istep in trait t
    --! | 1. (m=matarrmax(a, (i-1)))
    --! | 2. (FORALL ar, k::(matarrmax(ar, (k+1))=
                                          (IF (matarrmax(ar, k)>(ar[(k+1)]))
                                           THEN matarrmax(ar, k)
                                           ELSE (ar[(k+1)]))))
--! | >> (IF (i<=n)
          THEN (IF (m<(a[i]))
                 THEN ((a[i])=matarrmax(a, i))
                 ELSE (m=matarrmax(a, i)))
          ELSE (m=matarrmax(a, n)))
--! | <proof>
--! []
```

Let us instantiate the universally quantified variables ar and k on a and i respectively.

```
--! VC Status: ** not proved **
--! BY axiom istep in trait t
--! | BY analysis of FORALL, in 2 with a
--! | | BY analysis of FORALL, in 3 with i                    ...(1)
--! | | | 1. (m=matarrmax(a, (i-1)))
--! | | | 2. (FORALL ar, k::(matarrmax(ar, (k+1))=
                              (IF (matarrmax(ar, k)>(ar[(k+1)]))
                               THEN matarrmax(ar, k)
                               ELSE (ar[(k+1)]))))
--! | | | 3. (FORALL k::(matarrmax(a, (k+1))=
                          (IF (matarrmax(a, k)>(a[(k+1)]))
                           THEN matarrmax(a, k)
                           ELSE (a[(k+1)]))))
--! | | | 4. (matarrmax(a, (i+1))=(IF (matarrmax(a, i)>(a[(i+1)]))
                                  THEN matarrmax(a, i)
                                  ELSE (a[(i+1)])))
--! | | | >> (IF (i<=n)
              THEN (IF (m<(a[i]))
                     THEN ((a[i])=matarrmax(a, i))
                     ELSE (m=matarrmax(a, i)))
              ELSE (m=matarrmax(a, n)))
--! | | | <proof>
--! | | []
--! | []
--! []
```

Looking at hypotheses 1 and 4, and the conclusion, we see that we instantiated hypothesis 3 on i, whereas it should have been on i-1. Observe the i in the line (1) in the reproduction above. Rather than killing the whole of line (1) in the proof, and entering it again, we may simply highlight the i alone, delete it, and enter i-1. Pressing <return> now, or highlighting something else, ensures that the proof will be redone throughout with the new value. Anything entered by the user anywhere in the Penelope editor—in a program, proof, trait, invariant, etc.—may be altered in like fashion. In this case, we now have the following.

```
--! VC Status: ** not proved **
--! BY axiom istep in trait t
--! | BY analysis of FORALL, in 2 with a
--! | | BY analysis of FORALL, in 3 with (i-1)
--! | | | BY simplification
--! | | | | BY simplification
--! | | | | | 1. (m=matarrmax(a, (i-1)))
--! | | | | | 2. (FORALL ar, k::(matarrmax(ar, (k+1))=
                                (IF (matarrmax(ar, k)>(ar[(k+1)]))
                                 THEN matarrmax(ar, k)
                                 ELSE (ar[(k+1)]))))
--! | | | | | 3. (FORALL k::(matarrmax(a, (k+1))=
                           (IF (matarrmax(a, k)>(a[(k+1)]))
                            THEN matarrmax(a, k)
                            ELSE (a[(k+1)]))))
--! | | | | | 4. (matarrmax(a, i)=(IF (matarrmax(a, (i-1))>(a[i]))
                                   THEN matarrmax(a, (i-1))
                                   ELSE (a[i])))
--! | | | | | >> (IF (i<=n)
                  THEN (IF (m<(a[i]))
                        THEN ((a[i])=matarrmax(a, i))
                        ELSE (m=matarrmax(a, i)))
                  ELSE (m=matarrmax(a, n)))
--! | | | | | <proof>
--! | | | | []
--! | | | []
--! | | []
--! | []
--! []
```

We have added a couple of <simplify>s to clean up. The first step after the alteration was to try <SDVS-simplify>. This did not finish the proof, and left the goal a little messy looking. It usually simplifies, but occasionally as here it, as we say in the trade, "complifies". Thus we used the less powerful <simplify>. Unlike <SDVS-simplify>, two applications of <simplify> will sometimes do more than one.

Studying hypotheses 1 and 4, and the conclusion, we see that the "THEN" branch of the conclusion follows, and must be provable. However, the "ELSE" branch looks dubious. In fact, we are being asked to show what may be informally displayed as follows.

```
1. (m=matarrmax(a, (i-1)))
4. definition of matarrmax(a, i) in terms of matarrmax(a, (i-1))
>> (n<i) -> (m=matarrmax(a, n))
```

Unfortunately, as it stands it is impossible to prove. There may be many n such that n<i, and the conclusion does not follow. The immediate thought is: 'The invariant must be strengthened!'. A stronger invariant would add more information to hypothesis 1. Clearly what we need is that if n<i, then n=i-1. In the execution of the loop, n<i will occur only after the last execution of the body, and then indeed n=i-1. Correspondingly, the branch of the proof which we are discussing is that corresponding to when the boolean condition fails and we leave the loop. How then to add our extra information? Here is an important heuristic. Think of everything which we know to be true at the point at which the invariant is given, and conjoin to the invariant those facts which have some bearing on the problem. We can tidy up later to produce an elegant result, if desired. Since i is incremented by 1 each time through the loop, from the boolean condition we see that it will terminate when i=n+1. Thus, i<=n+1 can be conjoined to the invariant. To do this, we highlight the whole invariant, hold the right mouse button down and select

<conjoin>. This gives us the template for conjoining a new component with the invariant. Alternatively, the invariant could be altered crudely by killing and re-entering, or by getting to the end of it <ctrl-e> and extending the term. We then get the following annotated body of **arraymaxof**.

```
BEGIN
  --! VC Status: ** not proved **
  --! BY axiom istep in trait t
  --! | BY analysis of FORALL, in 2 with a --> selected term not a FORALL
  --! | | BY analysis of FORALL, in 3 with (i-1)
  --! | | | BY simplification
  --! | | | | BY simplification
  --! | | | | | 1. (m=matarrmax(a, (i-1)))
  --! | | | | | 2. (i<=(n+1))
  --! | | | | | 3. (FORALL ar, k::(matarrmax(ar, (k+1))=
                                (IF (matarrmax(ar, k)>(ar[(k+1)]))
                                 THEN matarrmax(ar, k)
                                 ELSE (ar[(k+1)]))))
  --! | | | | | 4. (FORALL k::(matarrmax((i-1), (k+1))=
                            (IF (matarrmax((i-1), k)>((i-1)[(k+1)]))
                             THEN matarrmax((i-1), k)
                             ELSE ((i-1)[(k+1)]))))
  --! | | | | | >> (IF (i<=n)
                    THEN (((i+1)<=(n+1)) AND
                          (IF (m<(a[i]))
                           THEN ((a[i])=matarrmax(a, i))
                           ELSE (m=matarrmax(a, i))))
                    ELSE (m=matarrmax(a, n)))
  --! | | | | | <proof>
  --! | | | | □
  --! | | | □
  --! | | □
  --! | □
  --! □
  WHILE (i<=n) LOOP
    --| INVARIANT  ((m=matarrmax(a, (i-1))) AND (i<=(n+1)));
    <statement>
    IF (m<a(i)) THEN
      m:=a(i);
    END IF;
    i:=(i+1);
  END LOOP;
  RETURN  m;
END arraymaxof;
```

Our larger invariant has added an hypothesis to the goal, and unfortunately the hypothesis numbers used in the applications of <forall-anal> are no longer correct. We can kill the bulk of the proof and re-do it, or use a slicker method: 1. Highlight the whole proof from the first <forall-anal> on. 2. Delete it but store it in clipped <ctrl-w> (or alternatively hold the middle mouse button down and select <cut-to-clipped>). 3. Inspect the goal to determine the correct hypothesis numbers. 4. Re-insert the stored proof segment <ctrl-y> (or hold the middle mouse button down and select <copy-from-clipped>). 5. Highlight the individual hypothesis numbers 2 and 3 in the proof steps, and replace them by the correct ones 3 and 4.

Our proof now becomes the following.

```
--! VC Status: ** not proved **
--! BY axiom istep in trait t
--! | BY analysis of FORALL, in 3 with a
--! | | BY analysis of FORALL, in 4 with (i-1)
--! | | | BY simplification
--! | | | | BY simplification
--! | | | | | 1. (m=matarrmax(a, (i-1)))
--! | | | | | 2. (i<=(n+1))
--! | | | | | 3. (FORALL ar, k::(matarrmax(ar, (k+1))=
                                    (IF (matarrmax(ar, k)>(ar[(k+1)]))
                                     THEN matarrmax(ar, k)
                                     ELSE (ar[(k+1)]))))
--! | | | | | 4. (FORALL k::(matarrmax(a, (k+1))=
                              (IF (matarrmax(a, k)>(a[(k+1)]))
                               THEN matarrmax(a, k)
                               ELSE (a[(k+1)]))))
```

```
--! | | | | | 5. (matarrmax(a, i)=
                  (IF (matarrmax(a, (i-1))>(a[i]))
                   THEN matarrmax(a, (i-1))
                   ELSE (a[i])))
--! | | | | | >> (IF (i<=n)
                 THEN (((i+1)<=(n+1)) AND
                       (IF (m<(a[i]))
                        THEN ((a[i])=matarrmax(a, i))
                        ELSE (m=matarrmax(a, i))))
                 ELSE (m=matarrmax(a, n)))
--! | | | | | <proof>
--! | | | | □
--! | | | □
--! | | □
--! | □
--! □
```

Next, we use <thinning> to remove the two universal hypotheses obtained from the axioms, as we have already obtained all that we need from them. Thinning cleans up the display and usually saves computation time.

```
--! VC Status: ** not proved **
--! BY axiom istep in trait t
--! | BY analysis of FORALL, in 3 with a
--! | | BY analysis of FORALL, in 4 with (i-1)
--! | | | BY simplification
--! | | | | BY simplification
--! | | | | | BY thinning 3, 4
--! | | | | | | 1. (m=matarrmax(a, (i-1)))
--! | | | | | | 2. (i<=(n+1))
--! | | | | | | 3. (matarrmax(a, i)=(IF (matarrmax(a, (i-1))>(a[i]))
                                     THEN matarrmax(a, (i-1))
                                     ELSE (a[i])))
--! | | | | | | >> (IF (i<=n)
                    THEN (((i+1)<=(n+1)) AND
                          (IF (m<(a[i]))
                                             THEN ((a[i])=matarrmax(a, i))
                                             ELSE (m=matarrmax(a, i))))
                                      ELSE (m=matarrmax(a, n)))
--! | | | | | | <proof>
--! | | | | | []
--! | | | | []
--! | | | []
--! | | []
--! | []
--! []
```

Optimistically, we hit this with <SDVS-simplify>, and the VC is indeed proved. We clean up the proof by going back and removing the two applications of <simplify>, and possibly <thinning> (using <ctrl-w>, <ctrl-k> and <ctrl-y> intelligently in that order). Our final proof is the following.

```
--! VC Status: proved
--! BY axiom istep in trait t
--! | BY analysis of FORALL, in 3 with a
--! | | BY analysis of FORALL, in 4 with (i-1)
--! | | | BY SDVS simplification
--! | | | | BY synthesis of TRUE
--! | | | []
--! | | []
--! | []
--! []
```

It remains for us to prove the main program VC. This is as follows.

```
--! VC Status: ** not proved **
--! 1. (n>=1)
--! >> (((a[1])=matarrmax(a, 1)) AND (2<=(n+1)))
--! <proof>
```

Clearly all we need is the base case of the definition of **matarrmax** from the axioms. Let us add that and apply <SDVS-simplify>. The final verified function is now shown.

```
 --| TRAIT t IS
--| INTRODUCES matarrmax: intarray, integer-> integer;
--| AXIOMS:
--| bc: (matarrmax(ar, 1)=(ar[1]));
--| istep: (matarrmax(ar, (k+1))=(IF (matarrmax(ar, k)>(ar[(k+1)]))
                                    THEN matarrmax(ar, k)
                                    ELSE (ar[(k+1)])));
--| END AXIOMS;
.
.

IS
  TYPE arrayn IS ARRAY(integer) OF integer;
  FUNCTION arraymaxof(a : IN arrayn; n : IN integer) RETURN integer
    --| WHERE
    --|      IN (n>=1);
    --|      RETURN x SUCH THAT (x=matarrmax(a, n));
    --| END WHERE;
    --! VC Status: proved
    --! BY axiom bc in trait t
    --! | BY analysis of FORALL, in 2 with a
    --! | | BY SDVS simplification
    --! | | | BY synthesis of TRUE
    --! | | []
    --! | []
    --! []

  IS
    m : integer := a(1);
    i : integer := 2;

  BEGIN
    --! VC Status: proved
    --! BY axiom istep in trait t
    --! | BY analysis of FORALL, in 3 with a
    --! | | BY analysis of FORALL, in 4 with (i-1)
    --! | | | BY SDVS simplification
    --! | | | | BY synthesis of TRUE
    --! | | | []
    --! | | []
    --! | []
    --! []
    WHILE (i<=n) LOOP
      --| INVARIANT  ((m=matarrmax(a, (i-1))) AND (i<=(n+1)));
      IF (m<a(i)) THEN
        m:=a(i);
      END IF;
      i:=(i+1);
    END LOOP;
    RETURN  m;
  END arraymaxof;
.
.
```

4.2 The factorial function

It behooves any document on program verification to treat the factorial function, and we now do this. Actually we will verify a different version in the later chapter on library units, and it will be useful to compare the two.

We will specify the function in the usual inductive manner, but implement the function iteratively. The recursive implementation is too close to the specification to be interesting. Here is our complete Penelope display with Ada code and specification, and tentative loop invariant. We have displayed the VCs.

```
--| TRAIT blah IS
--| INTRODUCES f: integer-> int*ger;
--| AXIOMS:
--| bc: (f(0)=1);
--| istep: ((k>=0)->(f((k+1))=((k+1)*f(k))));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
FUNCTION fact(n : IN integer) RETURN integer
  --| WHERE
  --|      IN (n>=0);
  --|      RETURN f(n);
  --| END WHERE;
  --! VC Status: ** not proved **
  --! 1. (n>=0)
  --! >> (1=f(0))
  --! <proof>

IS
  i, temp : integer := 1;

BEGIN
  --! VC Status: ** not proved **
  --! 1. (temp=f((i-1)))
  --! >> (IF (i<=n) THEN ((i*temp)=f(i)) ELSE (temp=f(n)))
  --! <proof>
  WHILE (i<=n) LOOP
    --| INVARIANT  (temp=f((i-1)));
    temp:=(i*temp);
    i:=(i+1);
  END LOOP;
  RETURN  temp;
END fact;
```

The main VC merely needs axiom bc and is no problem. The loop VC will need axiom istep, and presents two problems. The most obvious one is that the conclusion of the goal involves some facts about n, but the hypotheses don't mention n. As in our last example, we should add i<=n+1 to the invariant. The other problem is that in the conclusion an assertion is made about what is true when i<=n. Now i<=n includes the case where i<0, but the axioms only tell us about f for i>=0. Moreover, it includes the case where i=0, and this leads to similar problems. We need to add information to the invariant to tell the system that i>=1 always. This last shortcoming of the invariant is typical of the things which are often overlooked and usually discovered later in the proof when the user is wondering why it won't go through. These two extra properties have been conjoined, and the proofs given, below.

```
--| TRAIT blah IS
--| INTRODUCES f: integer-> integer;
--| AXIOMS:
--| bc: (f(0)=1);
--| istep: ((k>=0)->(f((k+1))=((k+1)*f(k))));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
FUNCTION fact(n : IN integer) RETURN integer
  --| WHERE
  --|      IN (n>=0);
  --|      RETURN f(n);
  --| END WHERE;
  --! VC Status: proved
  --! BY axiom bc in trait blah
  --! | BY SDVS simplification
  --! | | BY synthesis of TRUE
  --! | []
  --! []

IS
  i, temp : integer := 1;

BEGIN
  --! VC Status: proved
  --! BY axiom istep in trait blah
  --! | BY analysis of FORALL, in 4 with (i-1)
  --! | | BY SDVS simplification
  --! | | | BY synthesis of TRUE
  --! | | []
  --! | []
  --! []
  WHILE (i<=n) LOOP
    --| INVARIANT  (((temp=f((i-1))) AND (i>=1)) AND (i<=(n+1)));
    temp:=(i*temp);
    i:=(i+1);
  END LOOP;
  RETURN  temp;
END fact;
```

4.3 Automatic generation of invariants

When we select a loop to be added to the Ada code, the template is given with a place for the invariant. If now or later we highlight the whole loop and hold the right mouse button down we may select <compute-invariant>. This removes the user-entered invariant and the system then computes its own invariant, but does not display it, and produces the loop VC. This is a weak invariant which in all but trivial cases needs to be strengthened. It is strengthened by inserting an assertion (see the later chapter on Penelope features for a discussion of assertions) in the place where the invariant would normally go. This is not the most desirable way to deal with a loop. For details the reader is referred to [8].

4.4 Simple loop without iteration scheme

When writing Ada code, if we select <loop> we get the following template.

```
  --! VC Status: hidden
  --! □
  LOOP
    --| INVARIANT  <term>;
    <statement>
  END LOOP;
```

We must of course add an **exit** statement (discussed in a later chapter) ourselves if we wish to control non-exceptional termination. The invariant has the same function as for the `while` loop—roughly that it is supposed to be true every time control reaches the top of the loop body (regardless of where the `exit` statement is). We will not do any examples as familiarity with the `while` loop will enable the user to handle these.

Chapter 5

Specification of library units

This chapter is concerned with how the various library units of Ada relate to each other via their specifications in Penelope. In the current version of Penelope, the only library units which are dealt with are procedures, functions and packages.

Private types in packages, and with and use clauses are not implemented in the current version of Penelope. The principles of the the relationships between the various subprograms and packages can be understood by examining just subprograms, and so we confine ourselves to these for most of this chapter.

Each subprogram has its own specification, which must be justified by proof of the appropriate VCs. In Penelope, the specification of a subprogram is the only way in which it can relate to the outside world. For example, where a procedure call occurs in a program, predicate transformation across it must be based solely upon its specification, and will not involve the actual code in the procedure body given elsewhere.

The principles can be seen from the case of nested procedures. In the first section below, we show the logic involved with a procedure call.

5.1 Procedure calls

Suppose that our procedure has a global variable, or an IN OUT or OUT parameter—i.e. something which takes a value out of the procedure. The underlying logic uses a function derived from the procedure, which we refer to as the *skolem* function associated with the procedure, to represent the value of this object after the procedure has been executed. This function takes as parameters the initial values of all actual parameters and global variables of the procedure at the time of call. Each of the procedure globals and IN OUT and OUT parameters has its own such skolem function. We remark here that global variables are really a kind of hidden parameter, and that in Penelope we are required to declare them explicitly in the WHERE part, as can be seen below. The use of these skolem functions implicitly assumes determinism in the execution of Ada programs, which is a restriction. and thi will be changed in a later version of Penelope.

Abstractly, the logic of a procedure call is illustrated in the following pseudo-Penelope display. The actual display would not look like this. The skolem function for **x** is here written as **sk_x**, etc., but in true Penelope it has a long name indicating among other things exactly which procedure it comes from.

```
PROCEDURE main
  --| WHERE
  --|      IN inmain;
  --|      OUT outmain;
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS
  a, b, c, g : integer;
  PROCEDURE sub(x : IN OUT integer; y : IN integer; z : OUT integer)
    --| WHERE
    --|      GLOBAL g : IN OUT ;
    --|      IN insub;
    --|      OUT outsub(x, y, z, g);
```

```
      --| END WHERE;
      --! VC Status: hidden
      --! []

    IS

    BEGIN
      .
      .
    END sub;

BEGIN
  .
  .
  --: PRECONDITION =
      insub
      AND
      (
        outsub(sk_x(a, b, c, g), b, sk_z(a, b, c, g), sk_g(a, b, c, g))
      ->alpha(sk_x(a, b, c, g), b, sk_z(a, b, c, g), sk_g(a, b, c, g))
      );
  sub(a, b, c);
  --: PRECONDITION = alpha(a, b, c, g);
  .
  .
END main;
```

We have called the postcondition of the procedure call **alpha**. This has various variables free in it as shown. Since in the body of **main** all we know about **sub** is its specification, we must ensure that **insub** holds before the call, and hence its appearance as a conjunct in the precondition. For the same reason, we must ensure that **outsub->alpha** holds after the call. When **outsub->alpha** is transformed back across the call, we must replace all but the unchangeable **IN** variable, in the place of **y**, by the values they will assume after the call—given by means of the skolem functions. This explains the other conjunct in the precondition.

Skolem functions are large and cumbersome, and their appearance in VCs makes the VCs difficult to manage. It is, however, quite straightforward to avoid them. The following trivial program main calls the procedure incr to increment the value of variable a from 3 to 4. We have specified incr by saying "OUT ((y-IN y)=1)".

```
PROCEDURE main(a, b : IN OUT integer)
  --| WHERE
  --|     OUT (a=4);
  --| END WHERE;
  --! VC Status: ** not proved **
  --! 1. ((Func<"standard.main[predefined.integer,predefined.integer].
                incr[predefined.integer]", y>(3)-3)=1)
  --! >> (Func<"standard.main[predefined.integer,predefined.integer].
               incr[predefined.integer]", y>(3)=4)
  --! <proof>

IS

  PROCEDURE incr(y : IN OUT integer)
v    --| WHERE
    --|     OUT ((y-IN y)=1);
    --| END WHERE;
    --! VC Status: proved
    --! BY arithmetic

  IS


  BEGIN
    y:=(y+1);
  END incr;

BEGIN
  a:=3;
  --: PRECONDITION = (((Func<"standard.main[predefined.integer,predefined.
                              integer].incr[predefined.integer]", y>(a)-a)=1)
                              ->
                              (Func<"standard.main[predefined.integer,predefined.
                                   integer].incr[predefined.integer]", y>(a)=4));
  incr(a);
  --: PRECONDITION = (a=4);
END main;
```

The reader should examine the precondition to the procedure call incr(a) and observe how it arises from the previous logical illustration.

The y's in the *OUT* condition of incr, and the a's in the postcondition (a=4) must all be replaced by the skolem function for y, applied to the actual parameter a, during predicate transformation. The VC is perfectly provable, but is not intuitively clear to the user (especially if there are a few more skolem functions around). Penelope is designed so that if the first *OUT* condition is given in the form

```
   OUT y = ''some expression''
```

then ''some expression'' will be substituted for the actual parameter a throughout the rest of the predicate when it is transformed—thus avoiding the introduction of the skolem functions. Except in certain contexts, y's floating around in ''some expression'' cause the problem to remain, as these will generate skolem functions—and so this should be avoided. Rewriting the OUT condition in our example, we obtain the following.

```
PROCEDURE main(a, b : IN OUT integer)
  --| WHERE
  --|      OUT (a=4);
  --| END WHERE;
  --! VC Status: proved
  --! BY synthesis of TRUE

IS

  PROCEDURE incr(y : IN OUT integer)
    --| WHERE
    --|      OUT (y=(IN y+1));
    --| END WHERE;
    --! VC Status: proved
    --! BY synthesis of TRUE

  IS


  BEGIN
    y:=(y+1);
  END incr;
BEGIN
  a:=3;
  --: PRECONDITION = ((a+1)=4);
  incr(a);
  --: PRECONDITION = (a=4);
END main;
```

The OUT condition of incr is in the desired form. The occurrence of y in (IN y) on the right hand side gives no problems, as under predicate transformation, (IN a) becomes simply a before the call, with no need for skolem functions. The VC was proved automatically as in this form it was very simple.

Let us look at one more example.

```
--> 1 VCs NOT SHOWN!
PROCEDURE main(n : IN OUT integer)
  --| WHERE
  --|       OUT ((n MOD 2)=0);
  --|       OUT ((n-((3*IN n)*IN n))<=1);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  PROCEDURE evenup(x : IN OUT integer)
```

```
    --| WHERE
    --|      OUT ((x MOD 2)=0);
    --|      OUT ((x-IN x)<=1);
    --| END WHERE;
    --! VC Status: hidden
    --! []

  IS

  BEGIN
    IF ((x MOD 2)/=0) THEN
      x:=(x+1);
    END IF;
  END evenup;

BEGIN
  n:=((n*n)*3);
  evenup(n);
END main;
```

This program computes $3n^2$ and calls the procedure **evenup** to raise (if necessary) the result to the nearest even number. The main VC involves skolem functions, and it is our job to redo the specification of **evenup** in the aforementioned fashion. The cleanest way is to define a suitable function in a trait and define the OUT value of **x** using this function. This is done in the version below.

```
--> 2 VCs NOT SHOWN!
--| TRAIT blah IS
--| INTRODUCES f: integer-> integer;
--| AXIOMS:
--| def: (f(y)=(IF ((y MOD 2)=0) THEN y ELSE (y+1)));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
PROCEDURE main(n : IN OUT integer)
```

```
  --| WHERE
  --|      OUT ((n MOD 2)=0);
  --|      OUT ((n-(IN n*IN n))<=1);
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS

  PROCEDURE evenup(x : IN OUT integer)
    --| WHERE
    --|      OUT (x=f(IN x));
    --|      OUT ((x-IN x)<=1);
    --| END WHERE;
    --! VC Status: hidden
    --! []

  IS

  BEGIN
    IF ((x MOD 2)/=0) THEN
      x:=(x+1);
    END IF;
  END evenup;

BEGIN
  n:=(n*n);
  evenup(n);
END main;
```

We could avoid the use of a trait by using the "IF b THEN val1 ELSE val2"-style definition of x in the OUT secification of evenup, but the trait approach is best for more involved examples where we wish to control the appearance of the VC and the various subgoals.

For the multiple-parameter case, consider the abstract illustration at the start of this section. In the list of multiple OUT conditions, if the first OUT conditions of the procedure are given in the form

```
 OUT  x = ''some expression''
 OUT  z = ''some expression''
 OUT  g = ''some expression''
```

then the appropriate `''some expression''` will be substituted for a, c and g throughout the predicate

```
   (outsub(a, b, c, g) -> alpha(a, b, c, g))
```

in performing predicate transformation.

5.2 Function calls

Similar remarks to those made for procedures apply to functions. We will just give one example. The function `twicemin` calculates (mathematically speaking) $2 \times min\{x, y, z\}$, and in doing so calls a function `min` which calculates $min\{x, y, z\}$.

```
--> 2 VCs NOT SHOWN!
FUNCTION twicemin(x, y, z : IN integer) RETURN integer
  --| WHERE
  --|      IN incond;
  --|      OUT outcond;
  --|      RETURN somevalue;
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  FUNCTION min(u, v, w : IN integer) RETURN integer
    --| WHERE
    --|      IN (u>=0);
    --|      RETURN val SUCH THAT (((val<=u) AND (val<=v)) AND (val<=w));
    --|      RETURN val SUCH THAT (((val=u) OR (val=v)) OR (val=w));
    --| END WHERE;
    --! VC Status: hidden
    --! □

  IS
    lesser : integer := u;

  BEGIN
    IF (v<lesser) THEN
      lesser:=v;
    END IF;
    IF (w<lesser) THEN
      RETURN  w;
    ELSE
      RETURN  lesser;
    END IF;
  END min;
BEGIN
  RETURN  (2*min(x, y, z));
END twicemin;
```

The RETURN specification of min unwisely gives properties of the returned value val in expressions which involve val itself, and so skolem functions will be generated. We should instead enter a trait which defines a function—a mathematical version of min, say mathmin—which enables us to specify the returned value by

```
   RETURN mathmin(u, v, w)
```

using here the short form of the RETURN statement.

5.3 Declarations versus bodies

In Ada we can declare a subprogram in one place, and give the body in another. If this is done then in the current version of Penelope each gets the same specification. We enter the specifications in the declaration part and they automatically appear in the body. The next version, will have the possibility of different specifications with the obligation to show compatibility.

5.4 Recursive subprograms

Recursive subprogram calls are a special case of subprogram calls, which have been treated above.

We remark that often, with recursive procedures, skolem functions do not arise even though no care is taken to specify the procedure as described above. The reason is that the procedure is often called recursively right at the end of the defining text. Recalling the terminology of the above abstract description of the logic of procedure calls, this means that `outcond->alpha` is simply `outcond->outcond`, and despite the skolem functions, is reduced to `TRUE` automatically and the conjunct disappears from the predicate being transformed up through the program.

5.4.1 The factorial function revisited

We will give an example to illustrate the full use of the proving system, as above we were only interested in the logic of subprogram calls. In the previous chapter we verified an iterative version of the factorial function. Here we present the traditional recursive version. It is presented with the verification already completed, as we have already gone over most of what is involved in the building.

```
--| TRAIT blah IS
--| INTRODUCES f: integer-> integer;
--| AXIOMS:
```

```
--| bc: (f(0)=1);
--| istep: ((k>=0)->(f((k+1))=((k+1)*f(k))));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;
FUNCTION factrec(n : IN integer) RETURN integer
  --| WHERE
  --|      IN (n>=0);
  --|      RETURN f(n);
  --| END WHERE;
  --! VC Status: proved
  --! BY axiom bc in trait blah
  --! | BY axiom istep in trait blah
  --! | | BY analysis of FORALL, in 3 with (n-1)
  --! | | | BY SDVS simplification
  --! | | | | BY synthesis of TRUE
  --! | | | □
  --! | | □
  --! | □
  --! □

IS

BEGIN
  IF (n=0) THEN
    RETURN  1;
  ELSE
    RETURN  (n*factrec((n-1)));
  END IF;
END factrec;
```

5.5 Visibility and disambiguation

In Penelope we deal with annotated Ada programs. An annotation, such as a precondition, contains symbols corresponding to Ada identifiers, and mathematical symbols from traits (some, like +, predefined). The issues here are visibility and the resolving of ambiguity in annotations. For a full discussion of this, see [1].

The visibility of the symbols corresponding to Ada identifiers follows in the obvious way from the normal Ada visibility rules. The mathematical symbols have a flat visibility structure: they are all globally visible. In the current version of Penelope we can have multiple traits but only at the top level. These traits can have an arbitrary number of functions, axioms and lemmas in them. A future version of Penelope will introduce scoping of traits.

Penelope records the full names of Ada identifiers according to where they were declared (along with much other information, of course). In annotations, though, Penelope displays to the user only what is needed to make the display unambiguous at that point, preferring to use simple names if posible. If, however, there is a symbol in an annotation for which there is another annotation symbol or an Ada identifier nearby, such that simple names would be ambiguous, *expanded* names will be used. For an example, see the display below.

```
PROCEDURE main
  --| WHERE
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  b, c : boolean;
  PROCEDURE sub
    --| WHERE
    --|      GLOBAL c : IN ;
    --| END WHERE;
    --! VC Status: hidden
    --! □

  IS
    a, b, c : boolean;                                          ....(1)
    PROCEDURE subsub
```

```
      --| WHERE
      --|      GLOBAL a,b,c : IN ;
      --|      OUT (b=c);                                                ....(2)
      --| END WHERE;
      --! VC Status: ** not proved **
      --! >> (("standard.main[].sub[]":b)=c)
      --! <proof>

    IS
      --: PRECONDITION = (b=c);
      b : boolean;                                                       ....(3)
      --: PRECONDITION = (("standard.main[].sub[]":b)=c);

    BEGIN
      --: PRECONDITION = (("standard.main[].sub[]":b)=c);
      b:=a;                                                              ....(4)
      --: PRECONDITION = (("standard.main[].sub[]":b)=c);
    END subsub;
  BEGIN
    NULL;
  END sub;
BEGIN
  NULL;
END main;
```

Statement (4) in the procedure body subsub contains an occurrence of b which refers to the b which is declared locally in the declarative item (3). The b in the OUT condition (2) refers to the b in the declarative item (1). The postcondition of (4) is derived from the OUT condition, and the b in it refers to that declared in (1). Thus the b in (4) and the b in the pre- and postconditions refer to different things, and must be distinguished if the user is to be able to make sense of the display. Thus in the pre- and postconditions Penelope gives the full name

("standard.main[].sub[]":b)

of the `b` declared in `(1)`. This name roughly follows the Ada style of expanded names. The "`standard`" part is a result of the fact that all compilation units defined in a Penelope session occur within the predefined package called "`standard`", in keeping with the Ada rules. The square brackets in `main[]`, for example, contain the types of the parameters, though there are none in this simple case. This type information is used to resolve ambiguity in the case of overloading.

Penelope does not, in the current version, perform sufficient static semantic checking to do overload resolution. though this will be in a future version.

Chapter 6

Various Ada features

In this chapter, we will look at some of the programming constructs which are a part of Ada but not as standard as the usual constructs like those found in, say, Pascal. Since the standard treatments—e.g. Gries's book—do not cover these, we will discuss the way they are dealt with for verification.

6.1 Exit statements

The `exit` statement is a kind of restricted `goto` which takes control out of a loop. It is for those special occasions when catapulting out of a loop is intuitively clearer, or more efficient, than the alternative using only while loops. A typical use is to come out of a number of nested loops, where use of only `while` loops can result in contrived and complicated boolean conditions. The logic of the `exit` statement is quite simple, and it is sufficient to illustrate it with an abstract example along the above lines.

```
BEGIN
   .
   .
   .
   outer: WHILE outercond LOOP
      .
```

```
          .
          .

          WHILE innercond LOOP
            .
            .
            .
            --: PRECONDITION = (IF exitcond THEN alpha ELSE beta);
            EXIT outer WHEN exitcond;
            --: PRECONDITION = beta;
            .
            .
            .

          END LOOP;
          .
          .
          .
        END LOOP outer;
        --: PRECONDITION = alpha:
        .
        .
        .

      END;
```

No special care need be taken over this construct, as the automatic treatment is perfectly natural.

6.2 Exceptions

Penelope currently deals only with user-defined exceptions. The current version of Penelope does not require the user to declare exceptions; this will be remedied in the next version. To enter an exception handler at the end of the sequence of statements of a frame, we highlight a <statement> placeholder at the end of the sequence and hit <return> on the keyboard. This gives the desired template. In the current version of Penelope, a frame can be either a block statement, or a body of a subprogram or package (since task units and generic units are not implemented). To avoid numerous fine distinctions, we will assume that our exceptions are raised in, propagated through, and handled in subprograms.

6.2.1 Exceptions which are not propagated

Here we are concerned with the case where the exception is raised during the execution of a sequence of statements for which the innermost enclosing frame has a handler for the exception. If the exception is raised, and execution of the exception handler is completed, this is considered to be "normal" termination of our subprogram. The **raise** statement is rather like a **goto** statement under these circumstances, and the usual **IN**, **OUT** and **RETURN** conditions are adequate to specify the subprogram. Here is an abstract illustration of the logic.

```
PROCEDURE main
  --| WHERE
  --|       IN inmain;
  --|       OUT outmain;
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS

BEGIN
  .
  .
  --: PRECONDITION = topexcep;
  RAISE hell;
  --: PRECONDITION = postraise;
  .
  .
  --: PRECONDITION = outmain;
EXCEPTION
  WHEN hell =>
    --: PRECONDITION = topexcep;
    .
    .
    --: PRECONDITION = outmain;
END main;
```

6.2.2 Exceptions which are propagated

Here we are concerned with the case where the exception is raised during the execution of a sequence of statements for which the innermost enclosing frame does not have a handler for the exception. This is the case where the exception is propagated. We will first discuss the specification and verification of this subprogram. When an exception is raised under these circumstances, execution of the subprogram is abandoned, and it is considered "abnormal" termination. We need a separate part of the specification to deal with this. The exception may or may not be raised in a particular execution. The usual OUT or RETURN parts specify what must hold in those cases where *normal* termination occurs. To specify what must hold when *abnormal* termination occurs, we have four types of construct, which are used in whatever combination is needed to establish the desired properties.

There are three types of *propagation-constraint* and a *propagation-promise.* If we highlight the WHERE specification part of a subprogram and click on the right mouse button, we have the options <propagation-constraint> and <propagation-promise>. Selecting the first gives us the *weak* propagation-constraint, and leaving this highlighted gives us the option of switching to the *strong* or *exact* propagation-constraint. Assume that we have an exception called hell. All four types of specification are displayed here, where the user-supplied exception name and conditions are in lower case.

All four of them are shown here.

```
  --| WHERE
  --|       IN incond;
```

```
--|      OUT outcond;
--|      RAISE hell => IN inweak;                    (1)
--|      IN instrong => RAISE hell;                  (2)
--|      RAISE hell <=> IN inexact;                  (3)
--|      RAISE hell=> PROMISE prom;                  (4)
--| END WHERE;
```

We will now briefly describe each of these.

1. Weak propagation-constraint. This one states that if hell is propagated then inweak was true when the subprogram was invoked. This is be ensured by the underlying logic, by conjoining (IN inweak) with the precondition of the statement. There is no assumption that the initial truth of inweak ensures the raising of hell.

2. Strong propagation-constraint. This states that if instrong is true initially, and the subprogram terminates, then hell will be raised. This is ensured by conjoining (NOT IN instrong) with the postcondition of the whole subprogram. Note that our program logic is establishing partial correctness.

3. Exact propagation-constraint. This is the conjunction of the previous two, and can be achieved by using both separately.

4. Propagation-promise. This states that if hell is raised, then at the point where execution is abandoned, prom is true. This is ensured by conjoining prom with the precondition of the raise statement. Thus this type of specification is a kind of "out" condition for exceptional termination. Unlike the previous types, which deal with the relevant IN conditions, this specification deals with what effect the execution of the subprogram body has had up to the point where the exception was raised.

The final situation to consider is when a propagated exception is caught by a subprogram which has the appropriate handler. Here we have something like a goto, from somewhere inside a subprogram call, to the exception handler. The subprogram which was called and propagated the exception will have propagation specifications as above, and these are used in the construction of the VC of the outer subprogram with the handler. The reader may examine the logic of this and the above in a Penelope session by appropriate use of cutpoint assertions. Alternatively, the predicate transformers document, [8] may be consulted for full details.

Chapter 7

Other Penelope features

We have described many Penelope features already. In this chapter we give special mention to some which we have not yet encountered.

The first two are the assertion, and cut-point assertion facilities. Both of these are designed to enable the user to have some control over the preconditions as predicate transformation proceeds upwards through the program. Most important is the simplification of the precondition at intermediate points. By simplification, we mean a reduction either in size or in intuitive complexity, which usually go hand in hand. There are two benefits from this. One is that it modularizes the work of verifying a program—things are tidied up and proved locally rather than being collected into a single large VC at the top. The other is that there is a reduction of the total complexity of the VCs generated. As a simple illustration, suppose that we have a precondition `A AND B`. Predicates typically get larger as they undergo transformation, and so we may expect to have precondition `A' AND B'` later, where `A'` and `B'` are larger than `A` and `B` respectively. If we are able to remove the conjunct `B` now, it will require less work than dealing with `B'` later.

After that, we deal with the application of axioms and lemmas to intermediate preconditions. The aim of this is to simplify the preconditions for the same reasons as given above.

7.1 Assertions

If in a Penelope session we have a highlighted <statement> placeholder, and click on the right mouse button, we see a menu of options. Along with the usual Ada statements there is a number of Penelope features, one of which is the <assertion> option. The result of adding an assertion is that when predicate transformation passes that point, the assertion is conjoined with the current predicate, as in the following example.

```
.
.
x:=3;
--: PRECONDITION = (assertcond AND precond);
--| assertcond;
--: PRECONDITION = precond;
y:=x;
.
.
```

There is no obligation on the user to prove anything to justify his entering an assertion, as doing so can only make a stronger VC.

One use for assertions is to strengthen a loop invariant in the case where we have selected <compute-invariant> to cause Penelope to generate the invariant automatically. This is described in the loop chapter, in 4.3. Another use is for testing, and for adding now what we know will be there later, during the developmental stage of producing a program. For example, if we call a procedure which we have not finished specifying (leaving it with the trivial IN condition TRUE), then we may use an assertion to conjoin information to the precondition TRUE so as to enable us to test the code above.

The most important application is for the user to add information which he knows to be true at that point, to enable simplification (to be explained shortly) of the current precondition. One case is where the added fact may be buried or expressed differently in the current precondition, and the new exposed conjunct is convenient for simplification. Another case is where this fact, while true, will not be added to the precondition until predicate transformation has proceeded upwards some number of statements. It is this case which we demonstrate in the example below. Suppose we have an **if-then** statement with preconditions as follows.

```
.
.
--: PRECONDITION = (IF (x<=0) THEN ((y-x)>=y) ELSE ((y+x)>=y));
IF (x<=0) THEN
  --: PRECONDITION = ((y-x)>=y);
  x:=(-x);
END IF;
--: PRECONDITION = ((y+x)>=y);
.
.
```

If the "true" branch is taken, then we know that **x<=0** at the start, and that **x>=0** at the end of the branch. This information is not incorporated until predicate transformation passes the "if" line with the boolean condition in it. We may choose to add the assertion that **x<=0** at the top of the branch as shown below.

```
.
.
--: PRECONDITION = (IF (x<=0) THEN ((y-x)>=y) ELSE ((y+x)>=y));
IF (x<=0) THEN
  --: PRECONDITION = ((x<=0) AND ((y-x)>=y));
  --| (x<=0);
  --: PRECONDITION = ((y-x)>=y);
  x:=(-x);
END IF;
--: PRECONDITION = ((y+x)>=y);
.
.
```

Now we highlight the precondition at the top of the "true" branch and select <simplify-precondition>. This applies the SDVS simplifier to the precondition, and the result is as follows.

```
.
.
--: PRECONDITION = (IF (x<=0) THEN (x<=0) ELSE ((y+x)>=y));
IF (x<=0) THEN
  --: PRECONDITION = (x<=0);
  --: SIMPLIFIED PRECONDITION;
  --| (x<=0);
  --: PRECONDITION = ((y-x)>=y);
  x:=(-x);
END IF;
--: PRECONDITION = ((y+x)>=y);
.
.
```

We see the simplified precondition at the top of the branch, which makes the precondition of the whole if-then statement ripe for simplification. This is a rather trivial example. A more significant one would be an if-then-else statement where each branch had a considerable amount of code, and we add some known facts to each branch. A special case of this addition of extra facts occurs when we know that the current precondition is unnecessarily general given the IN condition. The next section deals with the most general way of dealing with this.

7.2 Cutpoint assertion

If at a highlighted statement placeholder we select <cutpoint>, we get to enter an assertion which completely *replaces* the precondition at that point. Naturally, there is an obligation on us to prove that the new assertion implies the old precondition. The logic of the construct is shown below. The cutpoint assertion added by the user is in line (1), indicated by the word ASSERT.

```
.
.
x:=3;
--: PRECONDITION = cutasscond;
--! VC Status: ** not proved **
--! 1. cutasscond
--! >> precond
--! <proof>
--| ASSERT cutasscond;                        (1)
--: PRECONDITION = precond;
y:=x;
.
.
```

This feature can be used for examining the logic involved with various constructs. For example, the precondition **precond** in the display above was produced in a Penelope session, using a cutpoint assertion in the elided code at the bottom. The user should *enter cutpoint assertions* in various positions if he wishes to examine the logic of some construct for himself—e.g. for a simple loop.

It can also be used while developing a program. We can work on a particular segment of code, without worrying what comes after it, by setting the desired postcondition, and ignoring the local VC obligation. A precondition could be entered at the top of the segment, and the local VC proved, to check the desired functionality of the segment. These additions can be removed after checking, if desired. Similarly, various "partial" pre- and post-conditions can be put to a segment one at a time, to check different aspects of functionality separately, all of this being done *in situ*.

We now come to the most important use of cutpoint assertions: incremental simplification. Predicate transformation will produce (roughly) the weakest precondition when it reaches the top of the program being verified. This will usually be strictly weaker than the **IN** condition. This is typically true at an intermediate point also. More precisely, an intermediate precondition will usually be more general than what is required. This gives us some scope for choosing a new precondition which is simpler than the current one, but which is still sufficiently general that when transformed up to the top, it includes the IN condition. For the reasons discussed at the beginning of this chapter, it is advantageous if we can do the replacement. We use the cutpoint assertion to do this, and have a local VC to prove, to justify the action.

In practice, the intuitive reasoning is often as follows. We see a large intermediate precondition which we believe to be messy and unnecessarily general. We look at the IN condition, and observe that when execution reaches our intermediate point, the concise condition A *will certainly hold*. Now we enter condition A as a cutpoint assertion.

Here is an example where we see that there is another predicate which is exactly equivalent to the current precondition, but much shorter. The last part of the code sets n to a rounded up to an even number. The OUT condition, seen as the final precondition has this (rounding up or down allowed) as a conjunct.

```
BEGIN
  .
  .
  --: PRECONDITION = (otherstuff AND
                      (IF ((a MOD 2)=0) THEN ((ABS (a-a))<=1)
                       ELSE ((((a+1) MOD 2)=0) AND ((ABS ((a+1)-a))<=1))));
  IF (NOT ((a MOD 2)=0)) THEN
    n:=(a+1);
  ELSE
    n:=a;
  END IF;
  --: PRECONDITION = ((((n MOD 2)=0) AND ((ABS (n-a))<=1)) AND otherstuff);
END main;
```

Our conjoined component has become equivalent to TRUE when transformed up to the beginning of the code segment. We would like to replace the whole precondition by simply **otherstuff**. Simplification is not powerful enough, and so we should tackle it with the theorem prover after a cutpoint assertion. The VC has been displayed, and the new precondition shown below.

```
BEGIN
  .
  .
  --: PRECONDITION = otherstuff
  --! VC Status: ** not proved **
  --! 1. otherstuff
  --! >> (IF ((a MOD 2)=0) THEN ((ABS (a-a))<=1)
          ELSE ((((a+1) MOD 2)=0) AND ((ABS ((a+1)-a))<=1)))
  --! <proof>
  --| ASSERT otherstuff;
  --: PRECONDITION = (otherstuff AND
                      (IF ((a MOD 2)=0) THEN ((ABS (a-a))<=1)
                       ELSE ((((a+1) MOD 2)=0) AND ((ABS ((a+1)-a))<=1))));
  IF (NOT ((a MOD 2)=0)) THEN
    n:=(a+1);
  ELSE
    n:=a;
  END IF;
  --: PRECONDITION = ((((n MOD 2)=0) AND ((ABS (n-a))<=1)) AND otherstuff);
END main;
```

7.3 Apply-axiom and Apply-lemma

If we highlight a precondition, we have the <apply-axiom> and <apply-lemma> options. These are devices to enable incremental simplification by bringing in axioms and lemmas from a trait. We will discuss only <apply-axiom> here, as <apply-lemma> is entirely similar.

Suppose we have an axiom expressed as a (universally quantified) equality with a certain expression to the left of the equality symbol. All free variables are of course assumed to be universally quantified. Next suppose that we have a precondition at some point which contains the aforementioned expression. The <apply-axiom> option allows us to replace the given expression throughout the precondition by the right hand side of the equality in the axiom. Due respect is paid to variable bindings of course. Consider the example below.

```
--| TRAIT t IS
--| INTRODUCES ave: integer, integer-> integer;
--| AXIOMS:
--| ax: (ave(x, y)=((x+y)/2));
--| END AXIOMS;
--| LEMMAS:
--| END LEMMAS;

PROCEDURE sub
   .
   .
IS

BEGIN
   .
   .
  --: PRECONDITION = (c=(a+(2*((a+b)/2))));
  --: USE AXIOM ax IN TRAIT t WITH x, y = a, b;
  --: PRECONDITION = (c=(a+(2*ave(a, b))));
   .
   .
END sub;
```

The precondition was originally as in the bottom precondition shown above. When asking for the *substitution*, we must *instantiate* the (implicitly universally quantified) variables x and y in the axiom with values, here chosen to be a and b respectively.

Bibliography

[1] Odyssey Research Associates. Larch/ada reference manual. Technical Report TR-17-8, Odyssey Research Associates, 1989.

[2] D.Gries. *The Science of Programming.* Springer-Verlag, 1981.

[3] E.Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, Princeton, 1979.

[4] C. Douglas Harper. Guide to the Penelope editor. Technical Report TR-17-9, Odyssey Research Associates, 1989.

[5] C. Douglas Harper. The logical foundations of Penelope. Technical Report TR-17-10, Odyssey Research Associates, 1989.

[6] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[7] T. Redmond. Simplifier description. Technical Report ATR-86A (8554)-2, Aerospace, November 1987.

[8] W.Polak. Predicate transformer semantics for Ada. Odyssey Research Associates internal document, 1988.